



收获，不止Oracle

梁敬彬 梁敬弘 著

打开惊喜之窗 翱翔意识天空

兄弟出品



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

收获，不止 Oracle

梁敬彬 梁敬弘 著

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

在这本书里读者将会跟随作者一同对 Oracle 数据库的相关知识进行梳理, 最终共同提炼出必须最先掌握的那部分知识, 无论你是数据库开发、管理、优化、设计人员, 还是从事 Java、C 的开发人员。接下来作者再将这部分知识中最实用的内容进一步提炼, 浓缩出最精华的部分, 分享给大家。这是二八现象的一次经典应用。

这部分知识就是 Oracle 的物理体系结构、逻辑体系结构、表、索引以及表连接五大部分。通过阅读这些章节, 读者将会在最短时间以以一种有史以来最轻松的方式, 完成对 Oracle 数据库的整体认识, 不仅能在工作中解决常规问题, 还能具备一定的设计和调优能力。相信通过这些章节的学习, 会给读者的 Oracle 学习带来极大的收获。

然而, 作者最希望看到的是: 让读者的收获, 不止 Oracle。

为达到此目的, 作者精心将全书分成了上下两篇, 刚才所描述的具体知识点体现在全书的上篇, 而在下篇中, 读者将通过各种精彩故事、生动案例, 体会到该如何学习和如何思考, 在意识的天空抛开束缚, 无拘无束、尽情飞翔。

在这里, 读者也许会有疑问, 前面说的有史以来最轻松的方式是一种什么样的方式呢?

还请亲爱的读者自己去揭晓谜底吧。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有, 侵权必究。

图书在版编目 (CIP) 数据

收获, 不止 Oracle / 梁敬彬, 梁敬弘著. —北京: 电子工业出版社, 2013.5
ISBN 978-7-121-20070-0

I. ①收… II. ①梁… ②梁… III. ①关系数据库系统 IV. ①TP311.138

中国版本图书馆 CIP 数据核字 (2013) 第 063004 号

策划编辑: 张月萍

责任编辑: 徐津平

印 刷: 三河市鑫金马印装有限公司

装 订: 三河市鑫金马印装有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16

印张: 31.25

字数: 750 千字

印 次: 2013 年 5 月第 1 次印刷

印 数: 4000 册 定价: 59.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线: (010) 88258888。

序

随着 IT 技术的蓬勃发展，相关技术书籍也大量涌现，这是令许多从事 IT 行业的人士十分高兴的事。但是，大多书籍都是就技术领域来谈技术，读起来难免有些枯燥或感到深奥。而我读这本书的内容时，禁不住有一种冲动，一个声音从心里喊出：原来技术书籍也可以这样让读者喜爱！

从来没有为哪一本书写过序，也从没有想过可以为一本书写序，因为心中明白自己没有这样的能力。但是，当我们公司年轻的数据库专家梁敬彬将他的书稿发给我参阅时，我被这本书深深吸引住了，更被他的一句肺腑之言感动了：“我写完这本书，有一种如释重负的感觉，因为我把自己认为对广大技术人员最有价值的东西分享给大家了，我知道很多人可以因此不再为技术所累！”我为我们公司有这样一位年轻的、优秀的、行业内知名的数据库专家感到十分自豪。他是用自己的行动践行着我们福富公司“敬天爱人、追求卓越”的核心价值观；他把自己对广大技术人员的爱通过这本书传递出去了。游于技艺，然心与道合，心中甚感愉悦。于是，认真学习为其作序。

《收获，不止 Oracle》这本书从思维方式和做事方式的高度来阐述技术，相信这样的书对 IT 从业人员是十分有益的。在这个知识爆炸的年代，技术不仅学不完，而且还在不断更新。可以说，具体某阶段技术知识的学习都是暂时的，而学思想、学做事的方法论才是永恒的。授人以鱼不如授人以渔，学会学习，学会总结，学会思考，才是最重要的。这本书将思想、方法、案例、故事完美地融合到技术中，这不仅需要作者具备精湛的知识、丰富的实战经验，更需有思想的高度提炼、方法论的总结、实际案例的组织和故事情节的构思。可以说，这本书是浩瀚技术海洋中的一颗璀璨明珠。翻阅过程中，更是惊喜不断，本书除了站在思想的高度将技术和方法论巧妙结合，弥补了业内技术书籍的遗憾外，其妙趣横生、新颖别致的风格也超越了我的想象。读者阅读全书仿佛置身课堂，别具一格的课堂风格让人耳目一新，饶有趣味的故事构思让人拍案叫绝！对于从事和即将从事 IT 相关工作的工程师们来说，遇到这本书应该是一种幸运、一个善缘！

作为公司层面的技术专家，梁敬彬服务的客户遍布全国各地，他能在十分忙碌的工作之余，把自己宝贵的工作经验撰写成 50 余万字的生动有趣的经典技术著作分享给广大技术人员，这是他的努力不亚于任何人的证明！我很高兴、也很欣慰地看到我们的员工有这样的胸怀和激情！

此外，此书的校验及补充完善是由拥有多项核心专利的知名专家、清华大学计算机系毕业的梁敬弘博士完成的。感慨两位专家联袂奉献之余，更感受到他们兄弟间默契配合所传递的爱的情谊。

岂曰无衣，与子同袍。此书必为 IT 行业带来一股清新之风，打造出 IT 书籍的一段传奇佳话，成就一部永不过时的经典力作！我相信，此书将为广大读者开启惊喜之心扉，让读者飞翔在思想的天空、遨游在方法论的海洋、采摘到技术的累累硕果，相信收获远不止 Oracle！

福富软件公司副董事长、总经理：杨林

别出心裁 另出蹊径

——与梁敬彬先生序

敬彬的新书就要出版，邀我写一点感受，于是就有了这一段文字。

我和敬彬相识是在 2010 年，那时我正在编辑《Oracle DBA 手记》一书，偶然被他发表在 ITPUB 论坛上的一篇文章所吸引，那篇文章的题目是《DBA 小故事之 SQL 诊断》，其内容清晰、行文引人，于是就和他约了那篇稿子加入到书中，期间邮件往来再到北京会面，就此熟识。

从当时的一篇文章到今天的一本书，我能够清晰地看到作者一以贯之的思考和叙述方式，这种积累与坚持也正是作者成长和成功的要素之一。

当时那篇文章的感受和今天这本书是类似的，作者能够用曲折的笔触将自己的经历真实生动地再现出来，并且带领读者一起经历一次思维的探索，这是属于他的独特风格。

作者在书中反复传达的核心观点是：Oracle 数据库看似艰深的原理实际上和生活中的基本常识并无二致。理解了这一层意思，就能够克服对于这门技艺的畏惧之心，此后的学习自然就能够顺风顺水。

诚然如此，我也经常和朋友们说，对于 Oracle 的很多艰深算法，如果由我们去深思熟虑，其结果都必然大致相同。类似 HASH 原理，布隆过滤等算法，理解了你就只觉得巧妙而不觉艰深。

现在梁老师就为我们寻找了一系列源于生活、循序渐进的学习路线，如果你能够细心领会，就会觉得这一门技艺实在是趣味横生。

盖国强 (eygle)
Oracle ACE 总监
云和恩墨创始人
ACOUG 创始人

名家推荐

敬彬兄这本书有着与市场上其他 Oracle 书籍与众不同的特点，他通过一个个精彩的小故事，串起 Oracle 的核心知识和优化方法论，并时刻强调学习和工作的意识，如何不被技术束缚，如何跳出技术，意识和方法真的很重要。相信读完本书，你的收获，绝对不止 Oracle！

丁俊 (dingjun123)

ITPUB Oracle 开发版资深版主

《剑破冰山——Oracle 开发艺术》副主编

通读本书，如醍醐灌顶，豁然开朗，本书从实战出发，出发于技术，而超脱于意识，回味无穷，作者拥有多年的 Oracle 应用和体系架构设计的经验，付出了不亚于任何人的努力，总结出众多独到的经验，不失为一本好书，为学习和使用 Oracle 的技术人员带来诸多益处。

傅祥文

福富软件公司运营总监

由梁敬彬、梁敬弘兄弟合作的《收获，不止 Oracle》一书问世了。这对学习和从事数据库相关事务的业内人员来说，是一件幸事。读一本技术方面的书，或修一门课程、听一个讲座，大凡可能有三方面的收获：掌握相关的知识，提高解决问题的能力，激发学习、探讨有关问题的兴趣和热情。这些可能的收获不在一个层面上，后两者更为可贵。梁敬彬梁敬弘兄弟的这本书恰恰给读者提供了这样的机会。当然，要有真收获，还要有真努力。

梁敬弘曾是我的学生，不仅学业专精，围棋也下得很不错，是一个真诚而聪明的小伙子，跟他的哥哥相比，内向一些。梁敬彬与弟弟相比更善于沟通和表达，是一个数据库方面的专家，也

是一个很好的教师。在此，预祝本书的出版获得成功，同时也祝兄弟二人在事业上不断取得新的成就。

黄连生
清华大学计算机系教授

曾经有 Oracle 的初学者问我，怎么开始学习 Oracle？那时候我的答案很简单：“去下载 Oracle 的在线文档，包括 Database Concept、Administrator’s Guide，然后开始学着做实验。”诚然，对学习技术而言，在线文档是一个不可多得的利器，但是，对于一个刚刚开始接触 Oracle 的人来说，要从枯燥的英文文档中去学习和理解 Oracle 的技术体系，也许有点勉为其难。就算是市面上众多的 Oracle 技术书籍，多数也是堆砌满了技术细节，随时可能吓跑初学者。

好在，现在梁敬彬先生通过自己在日常工作和培训中的磨练，把自己对 Oracle 技术的感悟，通过一个一个小故事，浅显而又形象地展现了出来。对于初学者来说，可以慢慢地在一个个小故事中去了解 Oracle 数据库。读完这本书，你也许会恍然大悟：“哦，原来 Oracle 是这样子的。”

罗海雄 (rollingpig)
ITPUB Oracle 管理版资深版主



梁敬彬，网名 wabjtam123，任 ITPUB 版主、ITPUB 社区专家、福建富士通公司数据库专家。参与编写过《剑破冰山——Oracle 开发艺术》、《DBA 手记 2》等技术书籍，多年从事电信相关行业工作，负责系统架构设计、优化、培训等工作，有着丰富的数据库管理、设计、开发、培训经验和电信行业经验。

梁敬弘，清华大学计算机系博士毕业，在计算机领域和金融领域皆有建树，拥有多项计算机相关核心专利技术的同时还拥有金融行业的 CFP 等高级认证。现就职于华夏银行总行。

前言



迈出崭新一步——本书特点及存在意义

0.1 当今时代，既是最好的也是最坏的

近年来，我深刻地体会到，如今这个时代对于绝大部分技术人员而言是幸福的时代，能在这个时代从事 IT 技术相关工作的技术人员应该很开心。因为比起老一辈技术人员，现今的技术人员几乎从来就不缺资料。除了可以在各技术平台的官方网站下载到详尽且准确的学习资料，他们还可以很容易地在 Google、百度上搜索到自己想要的答案，也可以很方便地在各种技术论坛上注册提问从而获取别人的帮助，其中最著名的就是 ITPUB 论坛，我本人也从中受益匪浅。

然而，这个时代对技术人员来说也是痛苦的时代。无论是传统的电信、金融、证券行业，还是新兴的互联网行业，我们都不难发现运维系统涉及的数据量、访问量及并发量都在以惊人的速度不断激增。重压之下，很多 IT 系统运行举步维艰，技术人员压力巨大，甚是痛苦。除了负载压力外，系统的复杂度也随着业务复杂度的增加而成指数级增加，让开发设计人员头昏脑胀，让故障诊断及系统维护的相关技术人员无从下手。

痛苦还不止于此，在这个时代，除了海量负载和高复杂度，还有让技术人员应接不暇而无所适从的各种新技术。且不讨论 IT 所有技术，仅以数据库为例，除了以 Oracle、DB2 等为代表的一系列传统关系型数据库外，还有内存数据库、列式数据库、以 HBase 为代表的分布式数据库等等，让人眼花缭乱。此外，除了各类纷繁技术的选型外，我们还要适应具体各个版本的不断更新，以 Oracle 数据库为例，从早期的 Oracle 5 版本到今天即将正式发布的 Oracle 12c，Oracle 每一次新版本发布都需要我们去学习和适应，投入大量的精力和时间。

这是幸福的时代，也是痛苦的时代。这是最好的时代，也是最坏的时代（如图 0-1 所示）。

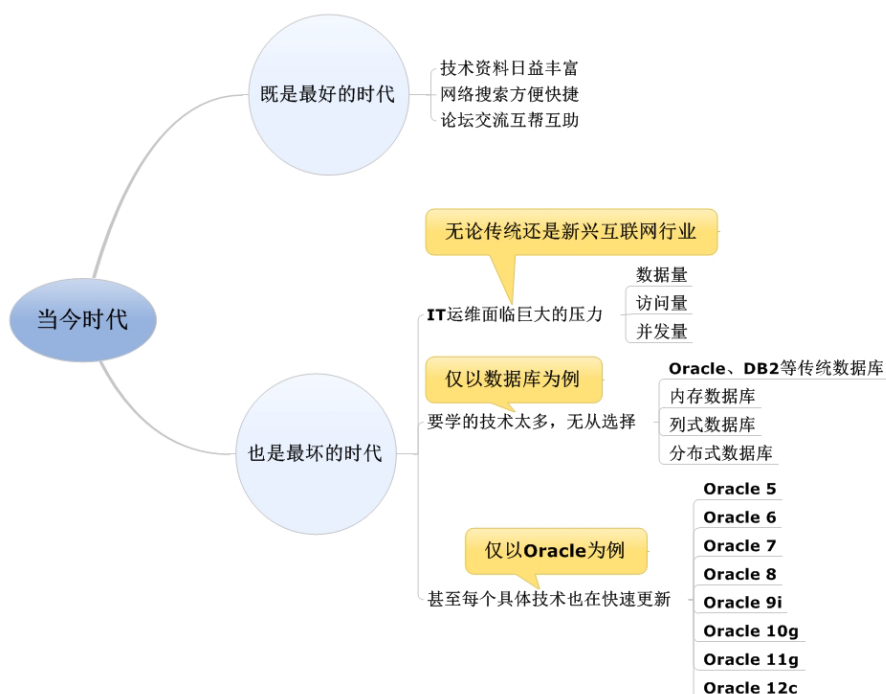


图 0-1 当今时代

0.2 技术人员，真正差距其实在意识

在这个特点鲜明的时代，我目睹了一大批在 IT 项目中不能有效适应这个时代而导致摔得鼻青脸肿，碰得一鼻子灰的技术人员，而这些人之中大多还是勤奋上进之人，少数还是技术能手。当然，也有成功的例子，让我来说说他们的故事吧。

0.2.1 小白的故事

工作两年多的小白是一个很努力的员工，每天除了用心工作外，还坚持充电学习，两年来他除了把 Oracle 10g 和 Oracle 11g 的官方文档几乎读了个遍，还购买了不少相关的书籍来学习。由于小白的项目组工作强度大，时常需要加班，因此小白工作学习外的时间就只够吃饭和睡觉了。那他的工作表现很出色吗，天道是否一定酬勤？实际情况却是不尽如人意，虽然他了解很多相关知识，可是在实际工作应用中却不得要领，频频出错，领导经常对其提出批评，他自己也相当苦恼。

其实小白只是因为缺少了一种称之为“意识”的东西，这个意识就是，该如何学习。在他的学习过程中，他犯了很多错误。

1. 忽略了知识的重点

小白读完了 Oracle 官方文档的几乎所有内容，毅力让人钦佩！遗憾的是，他忽略了二八现象这个事实：20%的知识，解决 80%的问题。小白根本不需要读完 Oracle 所有文档，他所在的项目组是从事数据库开发相关工作，而他却花费了大量的时间阅读完了至今他都尚未开始从事的数据库管理相关知识，比如备份恢复、RAC、DATAGUARD 部署及高级数据同步复制相关的技术，这是完全没有必要的。

另外语法方面的资料小白也根本无须细读、记忆，因为平时用得多的语法，自然记得住，遇到偶尔忘记的场合再去官方文档翻阅即可，现在 Google 等搜索引擎也非常强大，搜索到相关语法易如反掌。

人的精力是有限的，如果我们做到当前做什么事对应学习什么知识，尽量理解原理而不强记语法，将能省下多少宝贵的时间啊。

2. 从未考虑知识落地

小白曾经对我说过自己的烦恼，向我说明了他有多努力，看过了多少资料。我试探性地问他是否知道 Oracle 的体系架构是什么？他很自信地做了回答，滔滔不绝，甚至还画了草图给我，回答得让人相当满意。我又问他是否知道索引的结构和原理是什么？再次得到完美的答复。

接下来我再问他，知道这个体系架构对我们工作有啥帮助呢？他一下子回答不上来了。我继续问他，那索引的结构和原理对我们工作中的数据库应用有啥好处呢？再次答不上来。

原来这就是问题的关键所在！

知识要落地，要思考应用的场合，这就是我除了要让他把握学习重点外的第二个建议。他的学习其实就是为了学习而学习，没有思考应用场景的学习，是没有任何意义的。

后来我建议小白来听我在公司开讲的 Oracle 系列讲座，让他明白原来了解体系结构后我们居然可以将一条 SQL 从 42 秒调整到不足 0.01 秒；了解索引结构后，我们居然可以优化我们身边最为常见的 SQL 语句，比如 COUNT(*）、MAX()等。而在此之前，这些语句根本不会让他对索引产生联想。他终于彻底明白了什么叫知识落地，明白了意识有多么重要！

我最后强调，不只是 Oracle，其实学习任何技术都是一样的，没有思考过你所学的某项技术有什么用，没有想过如何落地，如何应用到实际工作中去，都是毫无意义的学习，纯粹浪费生命。

3. 选择技术使用场景

近来小白连续犯错，他学习了并行度这个章节后，知道并行可以有效地利用多个 CPU，就将自己的代码加入了并行度。但他忽略了生产系统不只运行他这一个应用这个事实，结果导致代码上生产后系统资源大量争用。直至最后我们定位到问题在他的代码中，将写法修正，取消了并行度后，系统终于恢复正常。

他为什么会犯这个错误？主要是因为他只专注于技术本身而忽略了应用的场景，如果是凌晨

2 点的某个大任务操作，他的这个设置就很好，因为那个时刻大多其他应用都已经停止运行了，资源不用白不用。

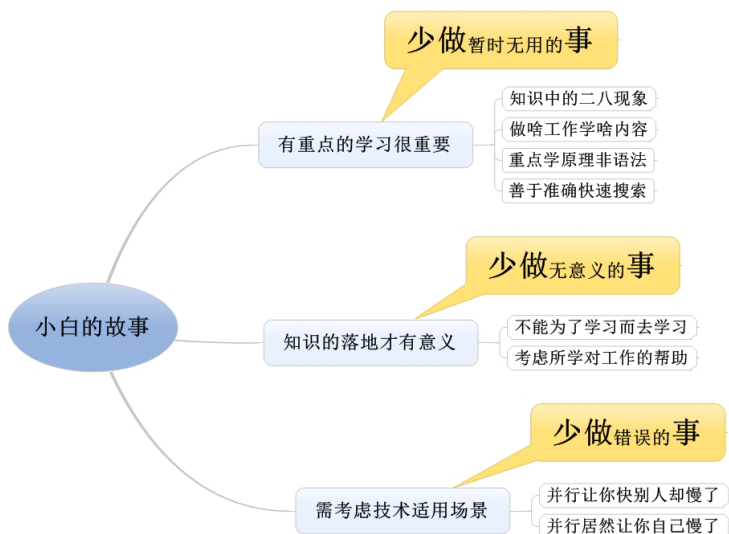
这就是什么时候选择什么技术。小白还有一个有趣的故事，就是在我给他提这第三个建议之后的一周，他又使用了并行，这次他不是在代码中写死，而是选择临时性场合使用，在凌晨 2 点运行系列特定脚本。结果这次运行的比想象的慢得多，还不如之前未用并行的时候。

大家知道是什么原因吗？其实还是因为不知道什么时候选择什么技术。这次他不会影响他人了，因为凌晨静悄悄。但是他影响了自己。因为他的任务的特点是小而多，平均不到 0.01 秒可以完成一条 SQL 执行。他忽略了另外一个细节，并行是需要调度的，调度是需要开销的，调度的开销甚至达到 0.1 秒，那当然是越跑越慢了。所以小任务实际上是不需要考虑并行的。

类似的故事还有很多，在本书中大家还会接触到什么时候选择什么技术的各类场景，这里暂且只说并行。

小白的故事说完了，其实要是早些年，他或许不会有这些苦恼，因为学习资料少，他即便不抓重点也不至于学到筋疲力尽；此外他的诸多失误都和性能有关，而早些年大多 IT 系统压力很小，在基本功能满足后性能问题几乎可以忽略不计，那小白也就没错可犯了。

再次强调，当今的 IT 建设项目，如果不对海量数据和高并发带来的性能问题进行有效的架构设计、部署规划，你的项目基本上就被判了死刑。而 Oracle 新版本为什么对应大量技术文档，很大一部分原因是 Oracle 新版本提供了更多的性能调优产品。为什么现在新技术如此繁多，其实也是源于此，难道内存数据库和列式数据及分布式数据库的产生，不是为了让系统跑的更快一些吗？只是这里要注意，它们不可能替代传统关系型数据库，因为它们都有各自适合的应用场合。小白的故事如图 0-2 所示。



0.2.2 小刘的故事

1. 该如何请教他人

工作近 5 年的小刘特别勤学好问，工作中一遇到不明白的技术问题就问，有时问周围的同事，有时发帖子在论坛上提问。可是最近他非常郁闷，因为总是没人愿意回答他。是不是大家不够团结友爱？非也！问题出在他自己身上，你们知道为什么吗？

让我们来看看他是怎么问的吧。

“请问，对表的某列建索引如何建？”

“请问，分区表中删除分区后，索引会不会失效？”

“哦，您说我问过同样的问题了，我怎么记不起来了？”

前两个提问有问题吗？不少人都觉得应该一点问题都没有，其实是很有问题的。因为很显然第一个问题可以 Google 搜索到，而第二个问题除了 Google 可以搜索外，你完全可以通过做一个试验来得出结论。

网上可以很方便搜索到的东西去问别人无异于浪费别人的时间，当今这个时代，搜索能力成为了最重要的能力之一。知道关键字的比不知道关键字的搜索能力更强；英文好的比英文糟糕的搜索能力更强。为什么不去锻炼提升自己这方面的能力呢？显然可以动手试验证明结论而不去试验，会让你白白失去了一个宝贵的试验和总结的机会。

第三句对话暗示该同学经常问同样的问题却不自知，这说明了什么？别人告诉你答案时，你思考过吗，记录过吗，总结过吗？

2. 问题该怎样描述

某次我在外地出差，下飞机后打开电脑，收到了小刘的一封求救邮件，内容是这样写的：今天早上上班发现宁夏的数据库很慢，请帮忙诊断分析一下，万分感谢！

之前我还收过小刘的另外一封求救邮件，内容是这样写的：紧急求救，湖北数据库崩了，请尽快帮忙处理，谢谢！

从这两封邮件可以看出来，求救者严重缺乏经验，在沟通上存在着巨大的问题，你们知道是什么问题吗？

先说这个慢的邮件，是某个局部慢还是整个系统都很慢，不同情况的处理方式截然不同。是今天开始忽然变慢还是一直以来都慢，直至今天再也无法忍受，不同情况的处理方式显然是截然不同的。再说这个慢字，有一次我对求助者说，这个语句 1 秒就出来了，你怎么会觉得慢得受不了？他的回答是，原先只需要 0.1 秒左右，这语句一天可是会跑上千万次啊。哦，原来 1 秒也让人觉得超级慢。还有一次有个求助者对我说，这个语句现在要跑 2 个多小时啊，原先超级快，只需要 5 分钟左右。哦，原来 5 分钟也有人觉得超级快。因此，你准确地告诉我现在要执行多长时间，希望执行多长时间，不是很准确吗？

再看第二封邮件，崩了？什么叫崩了，是数据库忽然关闭无法启动，还是系统运行太缓慢，还是数据文件丢失？哦，我来公布答案吧，后来知道这个崩了原来是指系统资源负载太高，系统

运行缓慢。原来这就是崩了，为什么要说这么模糊的字眼呢？

还有，这两封邮件都有个共同的特点，就是既没有提供接口人的联系电话，也没有提供所需要处理的机器的 IP 地址、登录用户名和密码。

毫无疑问，接下来我要再确认好多次，才能最终帮上他。

3. 失误不只是粗心

小刘最近非常苦恼，因为近期系统经常遇到并行相关的等待事件，导致系统运行缓慢，后来查出来是诸多表和索引都有默认的并行度，这是因为系统很多大表通过并行的方式完成了创建，小刘创建完毕后忘记将并行度取消了。因为此事影响了生产系统正常运行，小刘被领导批评了，他也承认了自己过于粗心的毛病。

其实我们仔细分析就知道，这种问题绝对不只是因为粗心。显然有个更重要、更本质的东西没有揪出来，这就是流程和规范。

如果小刘在操作前事先准备好操作步骤，而不是即兴发挥现场操作，取消并行度的步骤应该就不会忘记。

即便他忘记了这个步骤，连脚本都没有体现，也没有关系。如果有规范，要求生产的操作准备了脚本，并且必须要他人审核通过方可操作，那也就不会出错了。

即便审核的人也犯错了，如果有个完善的流程，在生产中完成操作后必须检查哪些项目，那个例行检查也必然会让错误再次避免。

人不是机器，都会失误，即便有的人素质非常高，不容易犯错，你也不能指望所有的人都这样，因此流程和规范才是最重要的。本书中大家会发现我们都有哪些流程和规范是用来避免误操作的。小刘的故事如图 0-3 所示。

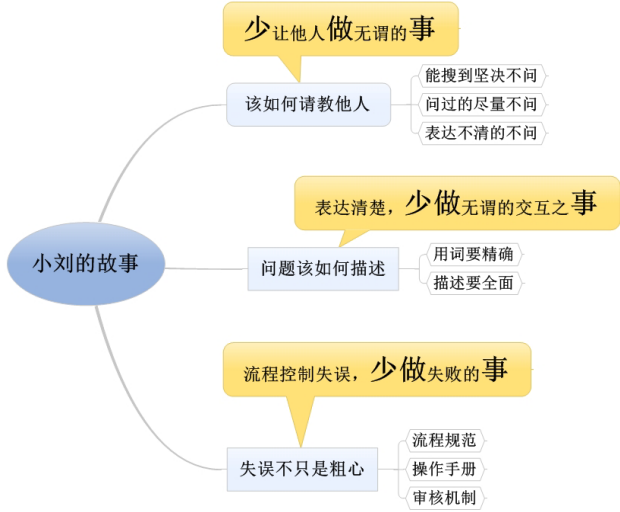


图 0-3 小刘的故事

0.2.3 自己的故事

1. 湖北神秘调优之旅

某日湖北某工程点向我求救，需要优化他们的生产系统，瓶颈正出在数据库上，为此公司让我去湖北现场进行调优。去之前我心中很没有底，因为据说这个性能问题已经困扰该工程点半年多了，期间该工程点已经请过好几波人马来调优过了，而且请来的人员中还不乏精通数据库的知名高手，这么多人这么长时间都无法把系统性能提升，我可以吗？

结果呢，我赶赴湖北现场忙碌一周后，该工程点生产系统性能最终得以大幅度提升，该工程点兴奋之余还发了一封表扬信到我公司，以此来肯定我的贡献。事后不少同事问我为什么那么多人长时间没解决的问题，而你一个人一周就搞定了，用了什么 Oracle 的高深技巧搞定这事。实际情况是，我并没有用任何 Oracle 技巧，甚至我根本就没有利用到和数据库有关的任何知识，但是我却解决了性能问题，圆满完成了任务。这是千真万确的事。

我到底做了什么事，让我来公布一下答案吧。我到了湖北，和现场技术支持人员以及相关人员对当前业务深入交流了一周，精简改写了大量逻辑被人为复杂化的 SQL，删除了部分多余的 SQL，对大表的保留记录情况根据业务特点进行了合理的规划，让不少大表变成了小表。

然后呢，然后就没有然后了，系统运行飞快，我胜利归来。这是一个很简单的道理，原先不少难以优化的运行缓慢的语句被我直接给咔嚓掉，从系统中消失了，此外不少大表变成了小表，系统变快不是自然而然的事情吗？

2. 三句话恢复的故障

某日早上 10 点，我接到项目组的紧急求救电话，被告知当天早上 8 点，某平台的生产系统运行极其缓慢，已经有相关人员在现场进行调查分析了，可惜两个小时过去了，直至现在依然没有解决问题，客户投诉不断，情况非常紧急。恰逢我在外地出差，且不说我在外地有重要事情要处理，即便是立即飞回来也需要一天的时间，该如何解决这个问题呢？

结果呢？10 分钟后，在我的帮助下，问题解决了，故障恢复了。我做了什么事，这么神奇？

其实，我只在电话里和求助者交流了短短的三句话，求助者心领神会而去，然后问题就解决了。我是让他们调整什么神秘的参数吗？不！其实我用的方法和数据库的知识一点关系都没有，他们要是早点给我打电话就好了，免得相关技术人员折腾了两个多小时，却无法消除故障，实际上这个技术人员的水平并不低，甚至可以说是 Oracle 方面的行家。

到底是哪三句话这么神奇呢？

“系统是从什么时候开始慢的，是一直就慢还是突然变慢？”回答是今早上班忽然感觉缓慢，昨天下班以前都是正常的。

“那昨天你做了什么吗？”回答是昨天晚上升级了补丁。

“补丁允许回退吗，如果允许，你就回退补丁吧。”回答是系统都不能用了，如果回退可以解

收获，不止 Oracle

决问题，当然可以回退。

接下来十分钟后，电话告知我，回退后系统正常了，问题解决了。

现在不影响生产系统的应用了，剩下的事情就是开发人员自己慢慢去检查代码有什么问题，最终问题在于代码有死循环和笛卡儿积，后续更改后投放生产就没问题了。

这其实就是生活，好比你去医院看病，肚子疼医生肯定会问你从什么时候开始疼，如果就是今天早上疼，那接下来必定会问你昨天晚上吃了什么……

3. 数据迁移有那么难吗

某日，某生产环境要做数据迁移，方法是通过 EXPDP 的方式将旧环境的数据库导出，然后通过 IMPDP 的方式导入到新环境，要求在凌晨 2 点开始导出，6 点前完成导入，因为之后需要测试，必须保证上班时间 8 点后，新系统可以正常运行。

这里涉及的技能是掌握 Oracle 的 EXPDP/IMPDP 命令，如果今天让我来上课描述这个工具的使用，大致 45 分钟时间可以完成课程的讲述，还包含了动手试验的时间。因此这显然不是一件很复杂的工作，当时现场的操作人员是一位工作多年的，拥有 OCM 证书的技术人员，大家也都觉得很放心。

然而实际情况是，从凌晨 2 点开始操作，直至下午 6 点，才完成了导入工作，整整比预计推迟了 12 个小时！这期间虽然临时做了不少应急补救措施，但是还是严重影响了生产的正常运营。

为什么会这么糟，中间出了什么问题？其实这里的故事太耐人寻味了，需要改进考虑的细节也太多太多了。

接下来揭晓谜底吧，还是通过一段对话展开。

“请问你要导出的库有多大？” 答曰不知道。

“请问你要导出的库最大的对象有多大，能否列举前 20 名的大对象？” 答曰没统计。

“请问是否是所有的对象都需要导出？” 答曰应该是，没确认。

“请问你知道导出和导入机器的 CPU、内存配置情况吗？” 答曰没注意。

OK，这就是原因了，让我来告诉大家我调查的结果吧，全库有 1TB 这么大，其中某记录操作日志的表 T1 就达到 400GB，连同索引，大小合计 500GB 左右，近乎一半的量。而和业务人员确认的结果是，T1 表可以暂且不导出，只需在新环境中建表结构即可。前 20 名的大对象除了巨无霸 T1 表外，还有不少也很庞大，其中有一张单表即有 120GB，只需保留最近三个月即可……经过我的确认，1T 的数据最终只需要导出 300GB 左右。

接下来的调查还发现了更加惊人之处，导出和导入的机器 CPU 个数居然都达到 64 个，强劲得让人惊叹！而我观察操作的脚本，也有写并行度，只是随便写了 PARALLEL=8。当时是凌晨，我们完全可以将导出导入的 PARALLEL 设置为更大，比如 60 个都不为过，这里又将会有 5 倍以上的提速。

后面我告诉他们，其实这次导出导入只需要合计 2 小时就足够了，不需要 4 小时，更不会操作了整整 16 个小时！

4. 充分准备是要领

某次我应邀去安徽某工程点进行数据库调优，周二晚上接到通知，要求周三下班前赶到现场，周四一天之内要把系统性能显著提升，从而应对周五集团组巡的检查工作，当时公司领导也在现场，情况相当紧急。

这是相当让人头疼的一件事，因为系统调优是一项相对复杂和艰巨的任务，要求在一天时间内立竿见影，可能性很小。虽说没有把握，但是也只好硬着头皮去试试看，我在出发前做了充足的准备，这些准备方法大多来自平时的积累，这种积累非常宝贵非常重要。有了这些重要的准备，我在出发前通过工程点现场同事的配合，获取了大量我了解的重要信息，在路上和飞机上做了充分的分析和研究，等下午 6 点到工程点现场，我已经对这里的情况了如指掌了。

然后，晚上调优到凌晨 2 点，第二天再奋战一天，第三天，系统终于运行平稳了，IDLE 从原先的 0~1%变为 40%~50%，系统性能显著提升了。

这个小故事说明了一个道理，积极的准备是很重要的，我准备了一个非常详细的处理及定位问题的流程和思路，从动态到静态，从整体到局部，非常详尽和实用，将会在后续分享给大家。读者可以从中体会到我是如何解决定位以及分析探索问题的，此外还可以学习到我所总结的优化思想和方法论。我自己的故事如图 0-4 所示。

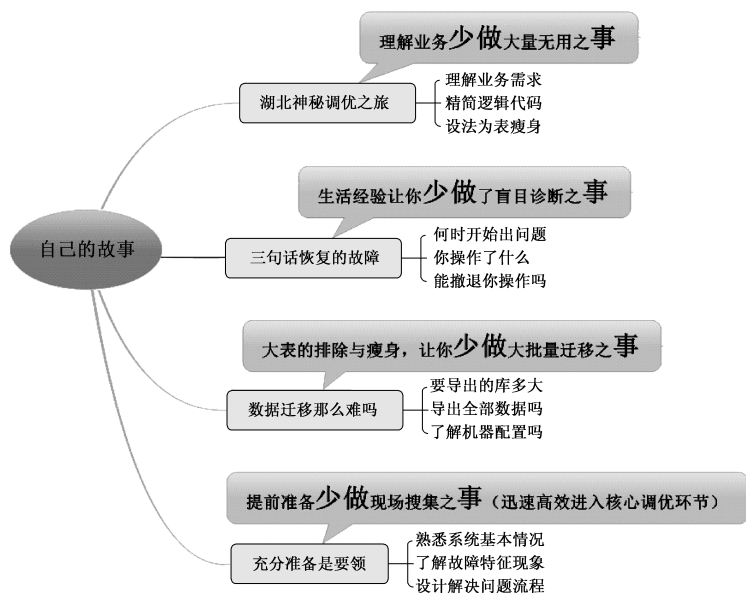


图 0-4 自己的故事

0.2.4 故事的总结

至此我讲完了小白、小刘和我自己的故事。别小看这三个人的故事，其实是非常典型的，这其中小白的故事告诉了我们学习是很有技巧的，是一门技术活。而小刘的故事告诉我们如何提问、求助以及制定规范是非常重要的。最后我本人的故事告诉大家，很多时候，解决问题不一定完全依靠技术本身，我的 4 个故事中的制胜法宝全部是非技术，这说明技术虽然重要，但是不能仅仅依赖技术。技术人员的故事如图 0-5 所示。

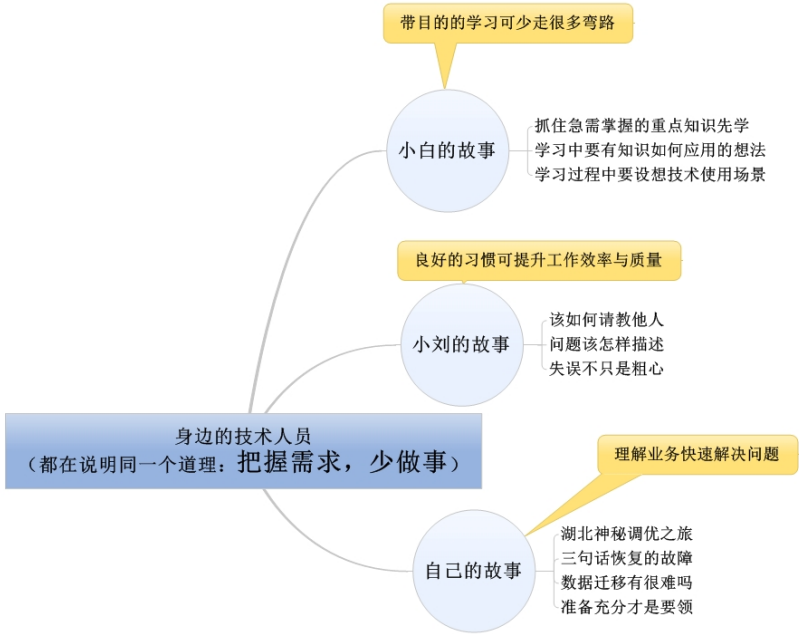


图 0-5 技术人员的故事

0.3 收获 Oracle，更收获“少做事”

包括我在内的几位技术人员的故事表面上说明的重点各不相同，实际上有一点上是相同的，那就是一个非常重要的意识：准确地把握需求，尽可能少做事！不要小看了这句话，这是永不过时的一句话，生活难道不是如此么？

精简起来可以用三个字概括：少做事。

就只是少做事？

别惊讶，让我们一起分析一下。

小白是一位数据库开发人员，数据库开发就是他的工作职责，很显然他的学习任务就是了解

数据库开发相关知识。他这时就应该选择性地学习，尽量抛弃和工作任务无关的知识，例如数据库的迁移备份部署等相关知识与他工作无关，就暂且不学，这就是少做事。

暂时在工作中应用不到的内容先不学，何乐而不为？接下来我们还要保证精简学习的内容，要掌握得最快，别拖拖拉拉地学了三五年，不符合这个时代的要求。这时最大的技巧就是将学习和工作结合在一起，带着知识点该如何落地应用以及什么时候该选择什么技术的想法去学习，这样目的性极强的探索性质的学习，往往学习一遍就够了，那些为了学习而学习的传统方法，学得多遍往往也无法有效地应用到工作中去。这又是少做事。

小刘为了更好地提升自己的技术水平，经常请教他人，希望快速获取别人的帮助，节省时间，本质是想少做事，这个出发点是对的。但是他问的问题是很方便可以搜到的，这样的提问对被提问者来说，就是让被提问者多做无意义的事，浪费生命，所以能搜索到的问题尽量不要去请教他人。同样道理，问过的问题也尽量不要再问。此外自己动手做试验来证明结论，这事表面上看不如直接问或者查资料来得快，不过试验可以让你在动手动脑的锻炼中加深印象从而过目不忘，和那些由于从不动手进行试验而导致学习过于浮躁、停留表面，最终反复学习多次都无法很好掌握相关知识的人相比，显然也是少做事了。

小刘的两封邮件表达得过于糟糕，导致我一头雾水无从下手，最终必然要再次联系反复交流。为什么不养成好的习惯，一次性说清楚，让大家都快速投入到故障处理工作中去？这也是少做事。

接下来说小刘的工作失误，文档规范、评审等看似是多做事了，其实也是典型的少做事，因为公司、部门的工作大多都有通用性，有了这些准备资料，交接的工作者可以直接拿来使用，既准确又省心，就是少做事。

最后点评自己的故事了。湖北之行是一次精简程序和瘦身表的旅程，显然是一次少做事的经典之旅，三句话解决故障显然是一次最精简的故障定位案例。接下来说数据迁移案例，从 1TB 到 300GB 导出，是否很有技术含量，可以说没有，也可以说有。说没有是因为这真的没什么技术可言，说有是因为有意识的人比起没意识的人，差别是非常大的。最后说说安徽之行我准备的文档和脚本，且不说提前准备后让我一到现场就开始解决问题，少做了定位问题的事（提前在路上完成了），节省了宝贵的时间，就说这份文档和脚本的分享，足以让后人少做不少事，节省不少时间。

点评完三个技术人员的故事，我的书要怎么写呢？前面说了这么久，其实主要在谈如何少做事的意识，现在这本书的主基调就明确下来了，就是主要和大家分享如何少做事解决问题的思路和方法。

分享这个主题首先要选择一个场景或者道具来和大家讲述，选择什么道具呢？考虑到现在 IT 行业中开发、设计、测试、维护等几乎所有 IT 相关技术员，或多或少都需要接触到关系型数据库，而 Oracle 又在当前关系型数据库的市场份额中遥遥领先，因此学习 Oracle 会让绝大多数人更容易找到合适的工作，从而在就业上多一些选择，少一些奔波。于是本着尽量少做事的原则，我们的

收获，不止 Oracle

道具确定是 Oracle。

围绕着 Oracle 课程我们将会惊奇地发现，其实 Oracle 技术的本身细节中，四处渗透着少做事的思想，从体系结构到物理结构，从表到索引的设计，无一例外。例如 SQL 执行第二次就会比第一次更快，显然是因为第一次执行保存了共享池的解析动作，CACHE 了磁盘中的数据，第二次无须再解析，无须再去物理磁盘读取数据，从而提升了性能，这就是少做事。又如索引的体积一般比表小得多，如果能只访问索引完成需求一定比在表中更快，这也是少做事。再比如分区表，如果没有设置分区表，则需要全表扫描，如果设置了，或许只需要在某个分区中查找即可，访问的路径少多了，这也是少做事……这些例子不胜枚举，多得可以写成一本书。哦，不对，就是可以写成一本书，这本书不就主要写这个吗？

本书不会将 Oracle 的所有知识都描述完，其中只涉及体系结构和物理结构，表、索引与表连接这几个章节，虽然涉及的内容不多，但是却是非常核心和重要的，所有不同角色的人员都离不开这几个章节的学习，这是共性的部分，只有把这几个章节学好了，后面的学习才有意义，这是为了避免大家像小白一样一头雾水地漫无目的地学习，少做和自己任务无关的事。另外最重要的是我讲述的学习的路线图和思路以及方法，这就是少做事的体现。

此外我特别强调知识的落地，探讨最多的话题就是，学习这些知识有什么用，和不知道这些知识的人相比，在工作中有什么差别，从而保证大家不浪费时间。在这些章节中，大家将会和我一起体会，原来理解体系结构后，我们可以把一条常见的 SQL 从原先的 42 秒完成提升到 0.01 秒左右完成，实现了一次从单车到飞船的飞跃；原来理解了索引后，立即就可以着手优化身边最常见的 SQL 语句。

接下来书中有另外一个重点部分，就是我的方法论、工作习惯、实用的准备工作脚本以及各个经典案例，这些章节再次强调了理解业务，尽量少做事不做事的思想有多么重要。

这就是本书的全部内容，学完这本书后，大家还会发现，原来用少做事的思想去学习数据库是那么的亲切，那么的容易理解和记忆。用尽可能将知识落地在工作中的意识去学习，学完后应用在工作中的那种感觉，是多么让人兴奋。

最后就是写书风格的确认了，这是一本风格非常有趣的书，灵感来源于我平时大量的授课。很多学员告诉我他们特别喜欢上我这样风格的课程，非常容易理解也非常实用。对此我很开心，猛然有一天有了这样的一个想法，将我的视频变成文字，这个想法得到我弟梁敬弘博士的大力支持，并答应帮我一起完成此书。于是全书就变成了“老师讲课、学生听课，老师提问、学生回答，学生提问、老师解答”这样的轻松对话方式。我终于实现了将视频变成文字。这本书成了一本轻松的书，读者很容易身临其境产生共鸣。而我讲述知识的方式也变得非常灵活了，有时故意不说的重点知识，在学生的思考中被引发出来，而读者也一同经历了这个难忘之旅。本书还有一个显著的特点，就是所有的知识点推论都有试验证明，而这些试验读者完全可以自己进行，再现老师的结果。这在保证课程准确性的同时，能让读者进一步加深印象。不过全书最有趣的特点还是

我构造的小余一家的故事。

学完本书后，大家就会发现，虽然当前时代数据压力比较大，但是并不可怕。其实调优如同生活一般，一点都不神秘。在老师的故事中，平均每处理 10 个成功案例中，就有 5 个不是依赖技术来解决的，真是一个有趣的数字。牢牢抓住需求，在设计和开发中巧妙地应用“少做事，不做事”的思想，产品就能高效，就很少存在性能问题，基本上就很少需要调优。

学完此书后相信大家会有一种强烈的感觉，原来学习技术可以如此轻松，原来学习要落地才有意义，原来少做事的意识这么重要。如果大家真有这个感觉，那我的所有努力——苦心积虑的书写风格、精心构造的试验脚本、精彩有趣的经典案例就没有白费了。我对于如何写此书的规划如图 0-6 所示。

最后我希望读者收获，不止 Oracle！

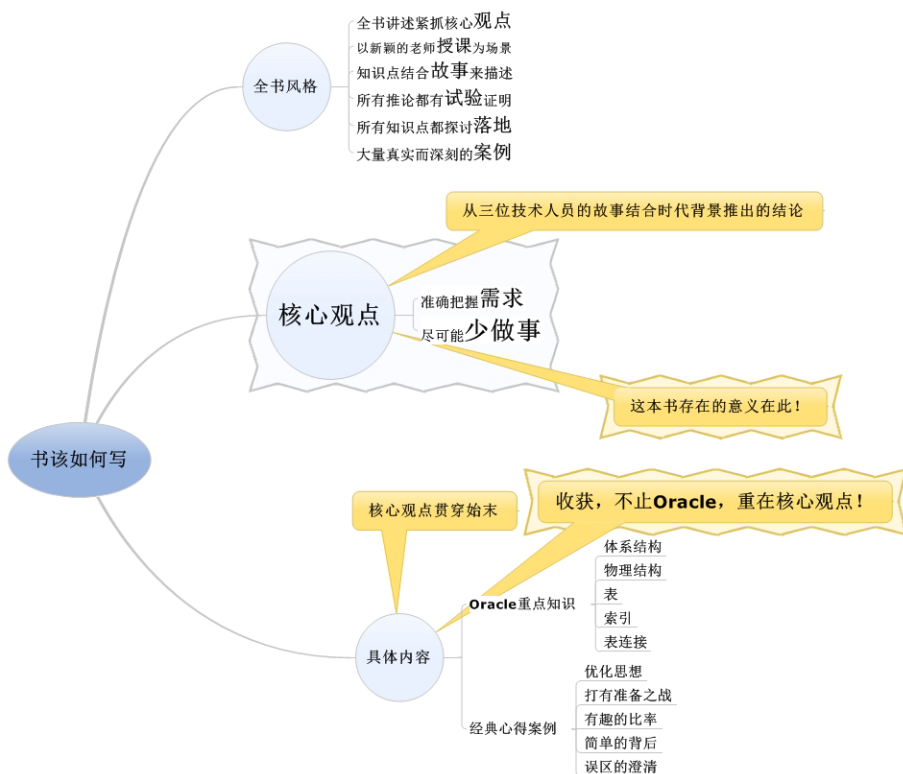


图 0-6 书该如何写

致 谢

我首先要感谢福富软件公司，因为公司文化所营造出来的这样一种氛围让我特别开心：各团队成员无论员工还是领导，都乐于分享、鼓励分享且勇于承担责任。

我至今忘不了当初姚建艺、郑清泉在我新融入团队时给予我的耐心指点和帮助，忘不了郑超群总监在我分享自己心得体会后发出的让我倍受激励的公开表扬邮件，也忘不了吴梦晟经理在我解决故障问题失利时只鼓励、不责怪、共同并肩战斗到天亮的场景。

正是在这种氛围的渲染下，才有了此书！

关于本书，我特别要感谢黄兴胜总助和傅祥文总监对我想法的鼓励、支持，以及对本书的关注、推进。尤其是要感谢我们的杨总，她在百忙之中读完我的全书手稿，提出宝贵意见并为本书写下热情洋溢的《序》，让我感动不已！

我很庆幸我有一个好弟弟梁敬弘，他从本书的最初策划、构思到中途的不断完善、补充，直至最终的反复审校，倾注了大量的精力。可以说，没有两兄弟的倾力合作，就不会有此书！

此外我要感谢苏旭晖、卢涛、丁俊、罗海雄、李丙洋这几位业内知名数据库专家对本书的审核校验，感谢博文视点在出版方面的专业性指导意见。感谢大家的帮助！

最后我要感谢我的爸、妈、妻、宝宝、弟、弟妹。谢谢你们的支持，我亲爱的家人！

梁敬彬

2013 年早春

目录



上篇 开启惊喜之门——带意识地学 Oracle

第 1 章 意识，少做事从学习开始.....	2
1.1 选择先学什么颇有学问.....	2
1.1.1 梁老师课堂爆笑开场.....	2
1.1.2 看似跑题的手机分类.....	4
1.1.3 学什么先了解做什么.....	5
1.2 善于规划分类才有效果.....	7
1.2.1 分类与角色密切相关.....	7
1.2.2 角色自我认识有讲究.....	9
1.3 明白学以致用方有意义.....	11
第 2 章 震惊，体验物理体系之旅.....	13
2.1 必须提及的系列知识.....	13
2.2 物理体系从老余开店慢慢铺开.....	16
2.2.1 老余的三个小故事.....	16
2.2.1.1 顾客的尺寸.....	16
2.2.1.2 有效的调整.....	17
2.2.1.3 记录的习惯.....	18
2.2.2 体系结构原理初探.....	20
2.2.2.1 从一普通查询 SQL 说起.....	20
2.2.2.2 老余故事终现用心良苦.....	23
2.2.2.3 一起体会 Oracle 代价.....	27
2.2.3 体系结构原理再探.....	30
2.2.3.1 从一普通更新语句说起.....	30
2.2.3.2 体系结构中提交的探讨.....	34
2.2.3.3 劳模的评选.....	38
2.2.3.4 回滚的研究.....	40
2.2.3.5 一致的查询.....	43
2.2.3.6 一致读的原理.....	46
2.2.3.7 实践的体会.....	49
2.3 体系学习让 SQL 性能提升千倍.....	65
2.3.1 一起探索体系学习的意义.....	65
2.3.1.1 同学们不知所学何用.....	66
2.3.1.2 实际上大有用武之地.....	67

2.3.2	单车到飞船的经典之旅	70
2.3.2.1	未优化前，单车速度	70
2.3.2.2	绑定变量，摩托速度	72
2.3.2.3	静态改写，汽车速度	74
2.3.2.4	批量提交，动车速度	75
2.3.2.5	集合写法，飞机速度	77
2.3.2.6	直接路径，火箭速度	78
2.3.2.7	并行设置，飞船速度	79
2.3.3	精彩的总结与课程展望	80
2.3.3.1	最大的收获应该是思想	80
2.3.3.2	老师的课程展望与规划	81
第 3 章	神奇，走进逻辑体系世界	84
3.1	长幼有序的逻辑体系	84
3.2	逻辑体系从老余养殖细细说起	85
3.2.1	农场之体系逻辑结构	85
3.2.2	农场之 BLOCK 漫谈	89
3.2.3	农场之区与段	91
3.2.4	农场之表空间的分类	93
3.2.4.1	表空间与系统农场	93
3.2.4.2	表空间与临时农场	93
3.2.4.3	表空间与回滚农场	94
3.2.5	逻辑结构之初次体会	94
3.2.5.1	逻辑结构之 BLOCK	94
3.2.5.2	逻辑结构之 TABLESPACE	95
3.2.5.3	逻辑结构之 USER	97
3.2.5.4	逻辑结构之 EXTENT	97
3.2.5.5	逻辑结构之 SEGMENT	98
3.2.6	逻辑结构之二次体会	100
3.2.6.1	BLOCK 的大小与调整	100
3.2.6.2	PCTFREE 参数与调整	101
3.2.6.3	PCTFREE 与生效范围	102
3.2.6.4	EXTENT 尺寸与调整	103
3.2.7	逻辑结构之三次体会	104
3.2.7.1	已用与未用表空间情况	104
3.2.7.2	表空间大小与自动扩展	105
3.2.7.3	回滚表空间新建与切换	109
3.2.7.4	临时表空间新建与切换	111
3.2.7.5	临时表空间组及其妙用	114
3.3	课程结束你给程序安上了翅膀	117

3.3.1 过度扩展与性能	117
3.3.2 PCTFREE 与性能	120
3.3.3 行迁移与优化	123
3.3.4 块的大小与应用	124
第 4 章 祝贺，表的设计成就英雄	131
4.1 表的设计之五朵金花	131
4.2 表的特性从老余一家展开描述	132
4.2.1 老余一家各施所长	132
4.2.2 普通堆表不足之处	132
4.2.2.1 表更新日志开销较大	133
4.2.2.2 delete 无法释放空间	136
4.2.2.3 表记录太大检索较慢	139
4.2.2.4 索引回表读开销很大	140
4.2.2.5 有序插入却难有序读出	143
4.2.3 奇特的全局临时表	146
4.2.3.1 分析全局临时表的类型	146
4.2.3.2 观察各类 DML 的 REDO 量	147
4.2.3.3 全局临时表两大重要特性	149
4.2.4 神通广大的分区表	153
4.2.4.1 分区表类型及原理	155
4.2.4.2 分区表最实用的特性	165
4.2.4.3 分区索引类型简述	176
4.2.4.4 分区表之相关陷阱	177
4.2.5 有趣的索引组织表	184
4.2.6 簇表的介绍及应用	187
4.3 理解表设计的你成为项目组英雄	189
第 5 章 惊叹，索引天地妙不可言	191
5.1 看似简单无趣的索引知识	191
5.2 索引探秘从小余缉凶拉开帷幕	192
5.2.1 BTREE 索引的精彩世界	192
5.2.1.1 BTREE 索引结构图展现	192
5.2.1.2 到底是物理还是逻辑结构	194
5.2.1.3 索引结构三大重要特点	198
5.2.1.4 插播小余缉凶精彩故事	201
5.2.1.5 妙用三特征之高度较低	203
5.2.1.6 巧用三特征之存储列值	219
5.2.1.7 活用三特征之索引有序	248
5.2.1.8 不可不说的主外键设计	265
5.2.1.9 组合索引高效设计要领	272

5.2.1.10 变换角度看索引的危害	289
5.2.1.11 如何合理控制索引数量	295
5.2.2 位图索引的玫瑰花之刺	297
5.2.2.1 统计条数奋勇夺冠	297
5.2.2.2 即席查询一骑绝尘	302
5.2.2.3 遭遇更新苦不堪言	306
5.2.2.4 重复度低一败涂地	309
5.2.2.5 了解结构真相大白	311
5.2.3 小心函数索引步步陷阱	315
5.2.3.1 列运算让索引失去作用	315
5.2.3.2 函数索引是这样应用的	317
5.2.3.3 避免列运算的经典案例	319
5.3 索引让一系列最熟悉的 SQL 飞起来了	325
第 6 章 经典，表的连接学以致用	327
6.1 表的连接之江南三剑客	327
6.2 三大类型从小余跳舞——道来	328
6.2.1 跳舞也能跳出连接类型	328
6.2.1.1 感觉怪异的嵌套循环	328
6.2.1.2 排序合并及哈希连接	329
6.2.2 各类连接访问次数差异	330
6.2.2.1 嵌套循环的表访问次数	330
6.2.2.2 哈希连接的表访问次数	337
6.2.2.3 排序合并的表访问次数	340
6.2.3 各类连接驱动顺序区别	341
6.2.3.1 嵌套循环的表驱动顺序	341
6.2.3.2 哈希连接的表驱动顺序	343
6.2.3.3 排序合并的表驱动顺序	345
6.2.4 各类连接排序情况分析	347
6.2.4.1 除嵌套循环都需排序	347
6.2.4.2 排序只需取部分字段	347
6.2.4.3 关于排序的经典案例	349
6.2.5 各类连接限制场景对比	350
6.2.5.1 哈希连接的限制	350
6.2.5.2 排序合并的限制	353
6.2.5.3 嵌套循环无限制	355
6.3 你动手装备的表连接威震三军	355
6.3.1 嵌套循环与索引	356
6.3.2 哈希连接与索引	362
6.3.3 排序合并与索引	363

下篇 飞翔意识天空——思想与案例的分享

第 7 章 搞定！不靠技术靠菜刀 368

7.1 SQL 被一刀剁了 369

7.2 整个模块丢弃了 370

7.3 调用次数减少了 371

7.4 排序不再需要了 372

7.5 大表砍成小表了 373

7.6 排重操作消失了 373

7.7 插入阻碍小多了 374

7.8 迁移事情不做了 375

第 8 章 升级！靠技术改隐形刀 377

8.1 大表等同小表了 378

8.2 大表切成小表了 379

8.3 索引变身小表了 380

8.4 删除动作不做了 380

8.5 清表角度变换了 381

8.6 提交次数缩减了 382

8.7 迁移越来越快了 384

8.8 SQL 语句精简了 385

第 9 章 提问，也是智慧的体现 391

9.1 描述要考虑周全 392

9.2 用词要尽量准确 393

9.3 说明要力求简洁 394

9.4 问过的避免再问 396

9.5 能搜能试不急问 396

第 10 章 买鱼，居然买出方法论 398

10.1 小余买鱼系列故事 398

10.1.1 诊断与改进 398

10.1.2 需求与设计 401

10.1.3 资源的利用 403

10.1.4 真正的需求 404

10.2 买鱼买出了方法论 405

10.2.1 一套流程 405

10.2.2 两大法宝 407

10.3 方法论的应用案例 408

10.3.1 从我们的这一套流程说起 408

10.3.1.1	诊断	408
10.3.1.2	改进优化（首次优化）	409
10.3.1.3	需求与设计（再次优化）	410
10.3.1.4	资源利用（花絮）	412
10.3.2	案例映衬了经典两大法宝	412
第 11 章	宝典，规范让你少做事	414
11.1	抓狂，为何事总忙不完	415
11.1.1	技术能力不足的新人们	415
11.1.2	不懂提问智慧的求助者	415
11.1.3	产生各种失误的粗心者	416
11.1.3.1	啊，小黄的 DDL 惹祸	416
11.1.3.2	惨，老师登错环境了	417
11.1.3.3	糟，小罗忘操作	417
11.1.4	解决问题缓慢的技术员	419
11.1.4.1	优化效率低下的小高	419
11.1.4.2	为何老师能快速解决	420
11.1.5	陷入种种困境的开发者	422
11.1.5.1	超长 SQL 使小郑烦恼	422
11.1.5.2	缺少注释让小叶沮丧	422
11.1.6	总是考虑不全的设计者	423
11.1.6.1	未提前规划的王工	423
11.1.6.2	不了解特性的刘工	424
11.2	淡定，规范少做无谓事	425
11.2.1	学习规范——促成新人快速成长	426
11.2.2	求助规范——引导求助不再迷糊	427
11.2.3	操作规范——协助粗心者不犯错	428
11.2.4	流程规范——保障问题快速解决	429
11.2.4.1	动态整体	429
11.2.4.2	动态局部	432
11.2.4.3	静态整体	439
11.2.4.4	静态局部	448
11.2.5	开发规范——让开发者驾轻就熟	451
11.2.5.1	SQL 编写规范	452
11.2.5.2	PL/SQL 编写规范	455
11.2.6	设计规范——助设计者运筹帷幄	457
11.2.6.1	表规范	458
11.2.6.2	索引规范	461
11.2.6.3	环境参数规范	467
11.2.6.4	命名规范	469

上篇



开启惊喜之门——带意识地学 Oracle

第 1 章 意识，少做事从学习开始

第 2 章 震惊，体验物理体系之旅

第 3 章 神奇，走进逻辑体系世界

第 4 章 祝贺，表的设计成就英雄

第 5 章 惊叹，索引天地妙不可言

第 6 章 经典，表的连接学以致用

第 1 章

意识，少做事从学习开始

本章的学习路线如图 1-1 所示。

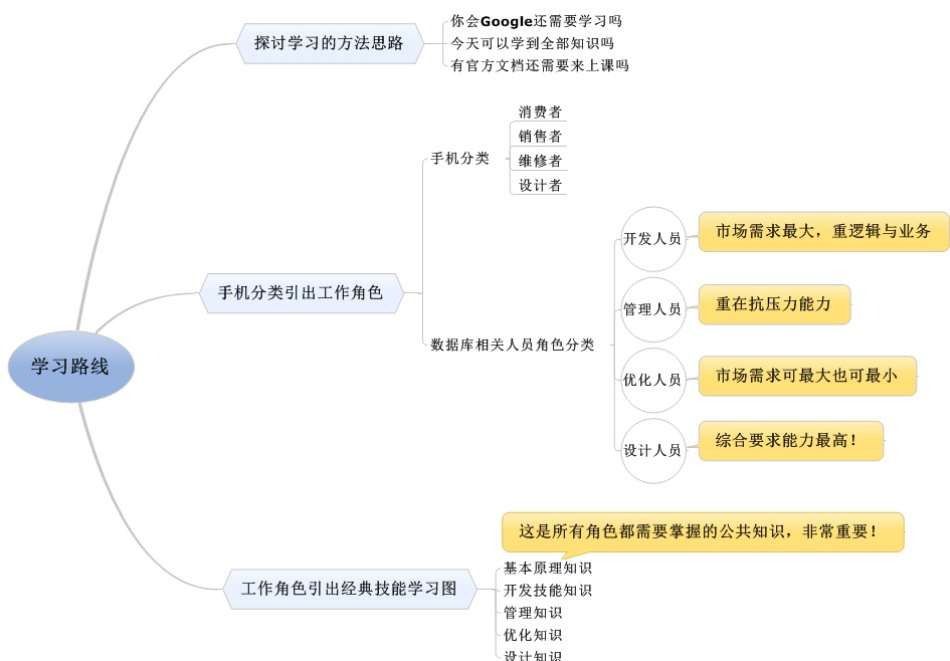


图 1-1 学习路线

1.1 选择先学什么颇有学问

1.1.1 梁老师课堂爆笑开场

公司新人培训开始了，小莲和众多新人一样，坐在培训大厅等候着梁老师的出现，今天的课程安排是 Oracle 数据库技能介绍。

之前已经培训 4 天了，大家先后学习了公共制度、公司业务、共通质量体系、企业文化、UNIX 基础知识……所有人都感觉很枯燥，也很疲惫，不过想到从明天开始公司会安排一周的新人野外拉练，心里倒是多了不少期待，大家都希望今天能过得快一点。

不一会儿，梁老师来了，准备好电脑和投影后，一天的课程即将开始了。清了清嗓子，梁老师的开场白开始了：

“各位同学，大家好！你们都是刚离开校园的新员工，你们中间接触过 Oracle 数据库的请举手。”

小莲犹豫了一会儿还是举手了，毕竟实习半年了，多少也接触过一些 Oracle 相关数据库环境。她转眼看周围，举手人数大致过半。

“看来还是有不少人没接触过 Oracle 数据库”，梁老师笑着说，“那有谁用过 Google 或者百度搜索工具的？”梁老师继续问。

这下大家都举手了，小莲放眼望去，没有不举手的同学。不过，接下来梁老师说的话，让大家谁也想不到……

“太好了，看来大家都懂得搜索，那你们后续工作中，遇到 Oracle 数据库相关应用的问题，有困难搜索就好了，今天我的课可以到此结束了，谢谢大家！”

台下沉默了半晌，接下来爆发出一阵笑声……

“大家笑什么啊？”梁老师微笑地问大家，“是不是可以下课了，很开心？”

场下继续笑……

“好了，大家都知道我是开玩笑，不过工作中遇到数据库相关的问题，不懂就搜，搜到就应用，所以我不用教大家了，这有啥问题吗？谁能提提反对意见？”

小莲觉得这个梁老师挺风趣的，不像先前那些老师那么死板，也就不觉得紧张了。积极开动脑筋后她随即举手发言了：“梁老师，我觉得如果什么都靠搜，那工作效率肯定会很低，应该是系统学习所有相关知识，遇到少数回忆不起来的内容时再搜索，这样才可以高效地完成工作。”

“回答得非常棒！”梁老师竖起大拇指表扬了小莲后继续问，“那你觉得今天一天可以系统地学完 Oracle 数据库的所有相关知识吗？”

“那肯定不行啊，我看书店相关的 Oracle 书籍都厚得超过一块大砖头，没个半年一年的，应该学不完吧。”小莲答道。

“既然一天不可能学完所有知识，最终还要靠大家自己平时学习，那还有没有人指望今天上完课后，立即变得精通数据库且在工作中驾轻就熟地解决各类数据库相关问题？”

场下摇头、微笑……

“下面，我会花 10 分钟左右的时间给大家指明一下学 Oracle 要读哪些官方文档，然后……”梁老师顿了顿，继续说道，“然后，大家就可以回家慢慢学习研究这些文档，今天的课就可以到此结束了。”

台下再次沉默了，接下来又爆发出一阵笑声……

1.1.2 看似跑题的手机分类

“看来大家对可以下课回家真是非常开心啊，我可以回去休息，也很开心，大家都开心，好事啊！”梁老师打趣地说道，“不过为了避免被公司开除了，俺还是要在这坚守一天。”

台下笑声不断……

“好了，别笑了，我们继续吧”，梁老师开始言归正传了。

“既然一天的课程安排不可能覆盖所有的 Oracle 知识，不能让你们立即就可以得心应手地工作，那我就干脆放慢上课的节奏，先和大家谈谈生活如何？”

场下气氛非常放松，大家都觉得今天非常轻松。

“其实学习数据库和生活经历是一样的，是相通的，那我和大家说说生活吧，大家应该都有手机吧，我就先从手机说起。”

“大家拿到新手机的时候，首先想了解或者说研究什么？”梁老师笑着问大家。

“研究怎样接打电话。”

“看看都有哪些功能。”

“看看能否发短信。”

“看看能不能上网。”

“看看能不能拍照。”

“看看能不能拍视频。”

“看看有啥游戏。”

……

“呵呵，回答得很热烈，非常好！大家回答了这么多，其实基本覆盖了手机的全部功能，我总结一下就是，大家首先最了解自己手中的新手机有哪些功能，怎么使用，并想快速熟悉它，对吧。”

“梁老师，你说的这个和 Oracle 技能培训有啥关系啊？”有些性急的小姑娘晶晶同学忍不住提问了。

“你觉得有关系吗？”

“梁老师，我觉得没关系啊。”

“没关系，也可以有关系。”

小莲瞥了一眼望去，看到晶晶一脸的迷茫，她也疑惑，这梁老师葫芦里卖的是啥药呢？

“我们继续吧，一部中档的智能手机，一般就会提供了非常多的功能，刚才大家七嘴八舌地也讨论了不少，那使用最多的、最常用的功能是什么呢？”

“接打电话！”台下几乎都是一致的回答。

“很好，那我再说说和手机有关的人群分类，我随意分类，将手机相关的人群分为消费使用者、销售者、维修者、生产设计者四类。大家觉得，如果我是这样分类，这些人群有啥明显的区别？”

“关注的重点不一样！”戴眼镜的小伙子曾祥大声回答。

“回答得太漂亮了！”梁老师满意地说道，“确实是关注的重点不一样！消费者关注的重点是手机的各个功能，如何使用；销售者关注的重点是价格、成本和市场；维修者关注的重点是故障的诊断定位；生产设计者关注的重点是手机原理、通信原理、行业技术标准、技术发展趋势等等。”

“现在，我们来做一个总结吧，关于手机我说了三点：

- ① 了解新手机有哪些功能；
- ② 讨论这些功能中哪些是最常用的；
- ③ 探讨与手机密切相关的人群如何分类。”

“跑题的梁老师，在 Oracle 培训课上和我们谈手机？”台下窃窃私语、议论纷纷……

1.1.3 学什么先了解做什么

“大家安静！现在我又开始说数据库了，欢迎回到 Oracle 培训现场。”梁老师停顿了一会儿，说道，“大家知道为什么我要说手机吗？”

摇头，还是摇头，台下半晌没人回应。

“因为我觉得学习数据库技术和各种生活经历是一样的，所以我才随意举了一个手机的例子。”

疑惑，还是疑惑，台下依然没人回应。

“大家都是刚开始接触数据库，好比大家刚刚接触一部新手机一样，面对数据库，你最想了解什么呢？或者说，你觉得数据库应该有哪些功能，请大家踊跃回答。”

“数据可以录入。”

“数据可以查询。”

“数据可以修改、删除。”

“不错，还有没有其他回答？”梁老师问。

半晌，没回音了……

“没了？如果真没了，那我不是来给你们上 Oracle 的，而是过来给你们上 Excel 的。好吧，请大家看屏幕，我新打开一个 Excel，接下来我们进行录入、查询、修改、删除……”

台下爆发出一阵笑声。

“笑什么？你们提到三点，不就是在描述 Excel 吗？其实数据库和 Excel 等最明显的差别在于，数据库是有事务的概念的。几笔不同的动作，如果在同一个事务里，要么一起成功，要么一起失

收获，不止 Oracle

败。举个场景来说明一下，好比你汇款转账 1000 元给你朋友，这里涉及两个动作，一个是从你的账户中扣款，另一个动作是在你朋友的账户中增加款，两个动作必须同时成功，否则就必须回退。假想汇款成功而收款失败了，整个事物却不回退，那你将平白无故地损失 1000 元。”

“好了，现在新增一个支持事务，还有没有其他的？”

台下依旧一片寂静……

“看来大家的思路没有被打开，我看还是继续说手机好一些，讨论手机的时候大家更踊跃一些。”这下不寂静了，一片笑声传来。

“这样吧，像刚才我们讨论和手机相关的人群分类一样，我们来聊聊数据库的分类，相信说完这个话题，大家思考数据库有哪些功能的思维就会更活跃一些。”

“手机的相关人群之所以可以分类，是因为他们关注点不一样。这里我也是针对关注点的不同对数据库进行分类的。我认为数据库应用可以分为数据库开发、数据库管理、数据库优化、数据库设计 4 类，各类侧重点如下。

- ① 开发：能利用 SQL 完成数据库的查增删改的基本操作；能用 PL/SQL 完成各类逻辑的实现。
- ② 管理：能完成数据库的安装、部署、参数调试、备份恢复、数据迁移等系统相关的工作；能完成分配用户、控制权限、表空间划分等管理相关工作；能进行故障定位、问题分析等数据库诊断修复相关工作。
- ③ 优化：在深入了解数据库的运行原理的基础上，利用各类工具及手段发现并解决数据库存在的性能问题，从而提升数据库运行效率，这个说着轻巧，其实很不容易。
- ④ 设计：深刻理解业务需求和数据库原理，合理高效地完成数据库模型的建设，设计出各类表及索引等数据库对象，让后续应用开发可以高效稳定。

好了，说到这里，大家在回答数据库功能的时候，还会只回答 4 点吗？”

“梁老师，我还能多说好多！”小莲听完这 4 个分类后，忍不住举手了。

“很好，我们先不回答这个问题了，这个作为培训后的作业吧，带回去思考然后将自己的想法反馈邮件给我。

现在大家和我一起分析思索了数据库的功能和基本分类后，大家也应该明白了我为什么要举手机的例子了，你们不觉得它们之间确实有相通之处么？其实这个相通本质是因为技术和生活是相通的。还记得我有问过大家手机的最主要的功能是什么吧，大家都回答是接打电话。为什么要这么问呢？那是因为生活和工作中，都存在二八现象，也就是百分之二十的功能实现百分之八十的需求。就以手机为例吧，一个传统的手机用户，百分之八十的需求就是来源于接打电话和收发短信。而手机提供的接打电话和收发短信功能只占了所有功能的百分之二十甚至更少。那对于比较传统的手机用户来说，他要花的精力是首先了解如何接打电话和收发短信，而不是把手机所有的功能都学个遍。

对于我们数据库学习而言，也是如此，如果你是一个开发人员，你就应该首先了解 SQL 和

PL/SQL 的编写，而不是数据库的备份与恢复。知识点非常多，但是我们要根据自己的工作性质来选择性地学习，这也是我将数据库进行分类的原因之一。手机销售人员首先要了解的是将要销售的手机的品牌、价位、成本、促销策略，而不是手机故障的定位诊断方法。

总结来说，我想表达的就是，你要了解数据库有哪些功能，数据库应用可以如何分类，并且要知道哪些是重点知识，是二八现象中的二，我们要先学习。这里需要大家用心去整理，去体会。”

结合梁老师所说的再回想起自己先前胡乱一把抓的学习方法，小莲猛然觉得有些顿悟，情不自禁地鼓起掌来，其他同学愣了一下，也都跟着鼓起掌来。

突然的掌声让梁老师发愣了一下，不过很快就会心地笑了。

1.2 善于规划分类才有效果

1.2.1 分类与角色密切相关

“前面说过了，Oracle 的知识点是非常多的，所以接下来我们要研究一个学习路线图，这才可以做到事半功倍。

按照我的习惯，我把 Oracle 知识分成如下 5 类，具体如图 1-2 所示。

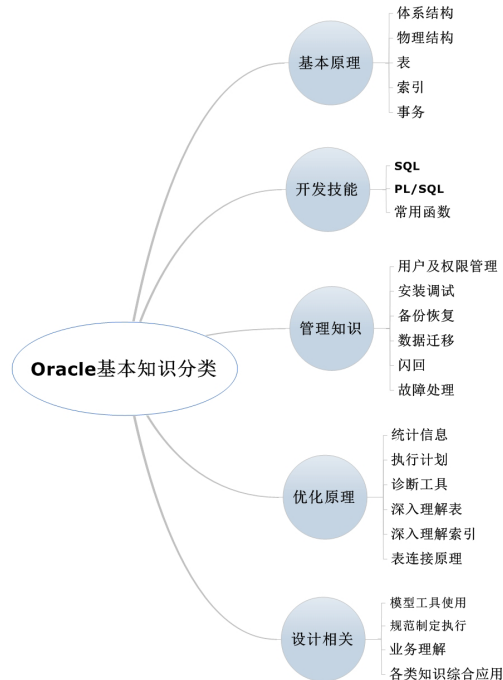


图 1-2 Oracle 知识分类

因此你首先确认当前工作的重心是什么，或者说你在数据库应用中的角色是什么，具体如图 1-3 所示。

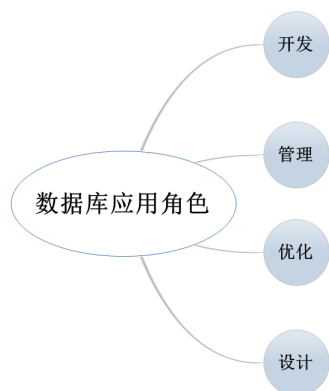


图 1-3 数据库应用角色

那不同应用角色，都要掌握哪些重点知识呢？结合上述所有分解图的分析，将输出如下一个非常重要的 Excel 图，注明了各个角色需要掌握的知识要点，如图 1-4 所示。

角色	基本原理					开发技能			管理知识						优化原理						设计相关			
	体系物理结构	体系逻辑结构	表	索引	事务	SQL	PL/SQL	常用函数	用户及权限管理	安装调试	备份恢复	数据迁移	闪回	故障处理	统计信息	执行计划	诊断工具	深入理解表	深入理解索引	表连接原理	模型工具使用	规范制定执行	业务理解	各类知识综合应用
开发	✓	✓	✓	✓	✓	✓	✓	✓								✓		✓	✓	✓				
管理	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓							
优化	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓				
设计	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

图 1-4 各角色需要掌握的知识要点

这时候应该有很多人关心这么多知识该如何获取，图 1-5 展现了知识获取的途径。

“哎呀，怎么觉得这会儿自己不是在上数据库技能培训课，而是在玩看图说话的游戏啊！”总结完前面的系列图片后，梁老师打趣地说道。

小莲扑哧一声笑了，在一片轻松的气氛下，梁老师让大家休息 15 分钟。

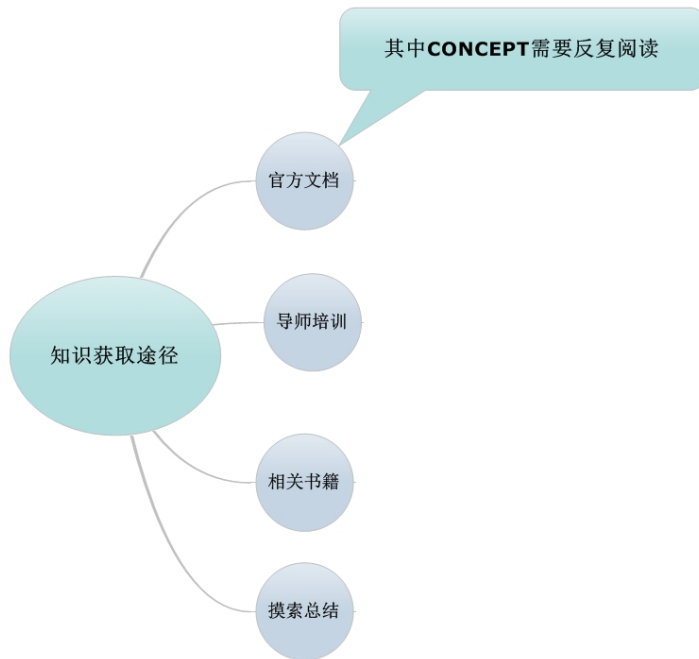


图 1-5 知识获取途径

1.2.2 角色自我认识有讲究

课间休息时间过得飞快，没一会儿梁老师开始召集大家坐回座位上课了。

“大家好，前面讲述的那个不同数据库应用角色所需掌握的知识要点，大家还记得吧？”听到大家肯定的回答后，梁老师开始发问，“大家能否说说我的这个知识要点图有啥特点？”

一小会儿寂静后，晶晶举手了：“梁老师，我发现无论是哪个数据库角色，基本原理都是必学的。”

“回答得非常好！”梁老师当即表示肯定。

“梁老师，我发现数据库各个角色中，设计人员最不容易，所有的知识点都需要掌握，而另外三个角色却都只是掌握部分即可。”胖乎乎的小伙子敬昱粗着嗓音回答道。

“完全正确！下面，我来详细描述这 4 个角色的特点。”梁老师继续说。

“从企业需求来说，数据库开发所需人数往往是最多的，学会了 SQL 和 PL/SQL 技巧后，大部分人直接就可以从事产品相关的开发，而无论哪个 IT 企业，从事产品开发的人员数量总是最多的。不仅是数据库开发，从事 Java 开发、C 语言开发的人员，只要他的应用和数据库有关系，就必然会应用到 SQL 和 PL/SQL，差别仅仅是多了一些封装工具而已。由此可见，数据库开发的学习是受众最广的，但是开发就离不开算法，对从业人员的逻辑思维能力要求比较高，尤其是面对

复杂的需求的时候。

相比数据库开发来说，数据库管理人员的人数需求在 IT 市场要少得多。这是由工作性质决定的。无论生产还是测试环境，搭建数据库都不可能非常频繁。另外如果数据崩溃需要恢复、数据需要迁移、紧急故障需要处理的情况频繁出现，那这个企业基本上也无法正常运营下去了。但是一旦出现问题，管理人员无法及时恢复故障，将会受到来自各方面的指责，压力非常大。和开发人员相比，管理人员不需要每时每刻地忙碌着，但是却要时刻注意充电，提升自己的应急处理能力，还需要时刻对系统进行健康检查，以防不测。此外，虽然开发在逻辑思维方面的要求要高于管理，但是责任和压力却远没有管理这么大。

接下来说数据库优化的角色，不少企业没有设置专门的数据库优化角色，它可能被融入资深开发、资深管理和资深设计人员的技能之中。对于有这样角色的企业来说，场景是这样的：生产环境运行缓慢，数据库管理人员通过跟踪诊断，查出问题所在，原来是系列 SQL 运行缓慢导致的整个数据库性能低下。那这个时候对于数据库管理人员来说，他的工作结束了，然后优化人员介入，利用自己的知识优化这些 SQL。在没有专门角色的场景下，可能是这个管理人员有着丰富的技能，他优化了这些 SQL，也可能是资深开发人员或者是资深设计人员优化了这些 SQL。但是从工作职责划分、从更专业的角度来说，应该设置专职人员。数据库优化所需要的人员是最难估算的，或许很多，或许很少，甚至没有，但是却是最重要的技能之一。其实开发、管理、设计三个角色，对优化相关知识的学习应该是多多益善，懂得越多，工作越得心应手。

最后说数据库设计，大家看我的角色技能图就知道了，设计需要掌握的知识点最多，从事数据库设计是最不容易的，这是属于核心岗位的位置，少数人的规划和部署决定了产品最终的质量和生命力。从市场需求来说，从事设计的人员最少。一般来说，一个应届毕业生在相关开发、管理岗位努力工作两年后，都可以把开发及管理工作的做得比较出色，优化工作的得心应手应该至少要三年以上，但是要想从事设计相关工作，一般需要五年以上的工作经验。不知道大家是否注意到，设计中已经强调了对业务的了解。”

台下听得津津有味，对于大多新人来说，这些都是他们没听过的，却是非常重要的。梁老师很清楚自己为什么要花费这么多精力来整理阐述这些东西，他工作十来年了，见过的技术人员来来去去、一波又一波。不少人工作多年后还没弄清楚自己手头的工作性质，需要学什么，要怎么去学，要学成什么样，或者说该先学什么，后学什么，一概不清晰，一片混乱。感叹之余，梁老师决定了这次新人培训的具体内容。

接下来，梁老师向大家提一个问题，“各位同学，在数据库相关的 4 个角色里，你们最愿意做什么呢？”

“开发。”

“管理。”

“设计。”

“优化。”

“设计和优化。”

“管理和优化。”

梁老师忍不住笑出来了，回答的内容还真丰富，连组合都有了！

“很好！大家知道我为什么要问这个问题吗？除了想了解一下大家各自的兴趣外，最重要的是为了告诉大家一个就业的技巧。很多人眼高手低，一毕业就想从事设计及优化相关工作，结果碰了一鼻子灰找不到工作，因为企业根本不给你这个机会。也有人一个劲地想做数据库管理工作，但是由于管理相关的岗位比较少，结果成功的人比较少。很多时候当兴趣和工作不匹配时，不要强求，要耐心找机会。比如 SQL 开发技巧掌握后，可以匹配到很多适合自己的岗位，轻易地获取工作机会，而精通 SQL 及 PL/SQL 开发技巧，对管理优化和设计都是有用的！这是多好的起步啊。

我觉得如果这样走过，应该是一个完美的路线。刚毕业从事数据库开发相关工作，后续有机会再从事管理相关工作，期间兼顾优化相关的技能学习，主动承担起优化的任务，争取成为一个兼职或者专职的优化人员。最后，随着各项技能的熟练掌握和业务知识的不断熟悉，在学习掌握了设计相关的知识后，水到渠成地从事数据库设计相关工作，在这相对较长的时间里，人来人往潮起潮落，你总能等到你感兴趣的工作角色。我觉得这样走过来的技术人员应该不是一般的技术人员，而是一个技术专家，企业的核心技术骨干。当然，大家千万不要有误解，就是认为设计就一定比管理好，管理就一定强过开发，市场的供需决定了人员的比例，但是各个岗位都可以有出色的专家，最完美的还是在自己感兴趣的领域中大展手脚！

今天，我给大家说了和数据库相关的工作都有哪些角色，他们分别需要掌握什么样的技能，并且说明了各个角色的差异，另外还希望大家对角色的供需市场有所了解，并提供了一个我自己认为完美的规划路线，希望我说的这些能对大家有帮助。”

台下响起了长久的掌声……

1.3 明白学以致用方有意义

“下面听老师来一段有趣的对话吧。

‘老师，索引这个章节我学完了。’

‘那你有什么收获啊？’

‘收获很大，我知道索引的原理是这样这样这样的……’

‘你知道索引如何使用吗？’

‘当然知道啊，在列上建索引，就可以检索数据更快啊。’

‘那你知什么场合下使用索引最有效吗？’

‘哦，我这倒没想过……’

收获，不止 Oracle

‘对了，之前让你首先阅读 Oracle 体系结构，都读完了吗？’

‘读完了。’

‘都能理解吗？’

‘可以理解，我知道体系结构是什么样的，是这样这样这样的……’

‘你知道理解这些体系结构在工作有什么用吗？’

‘不是明白原理就可以了吗，学习体系结构还要去想有什么用？’

‘真没想过吗？’

‘没有啊。’

OK，这段对话结束了，学什么没有想过如何用，这样的学习无异于浪费时间，白白多做了无用的事。请同学们记住这一小段对话，在后面的学习中的，大家会有所顿悟的。”

第 2 章



震惊，体验物理体系之旅

2.1 必须提及的系列知识

本章知识体系如图 2-1 所示。

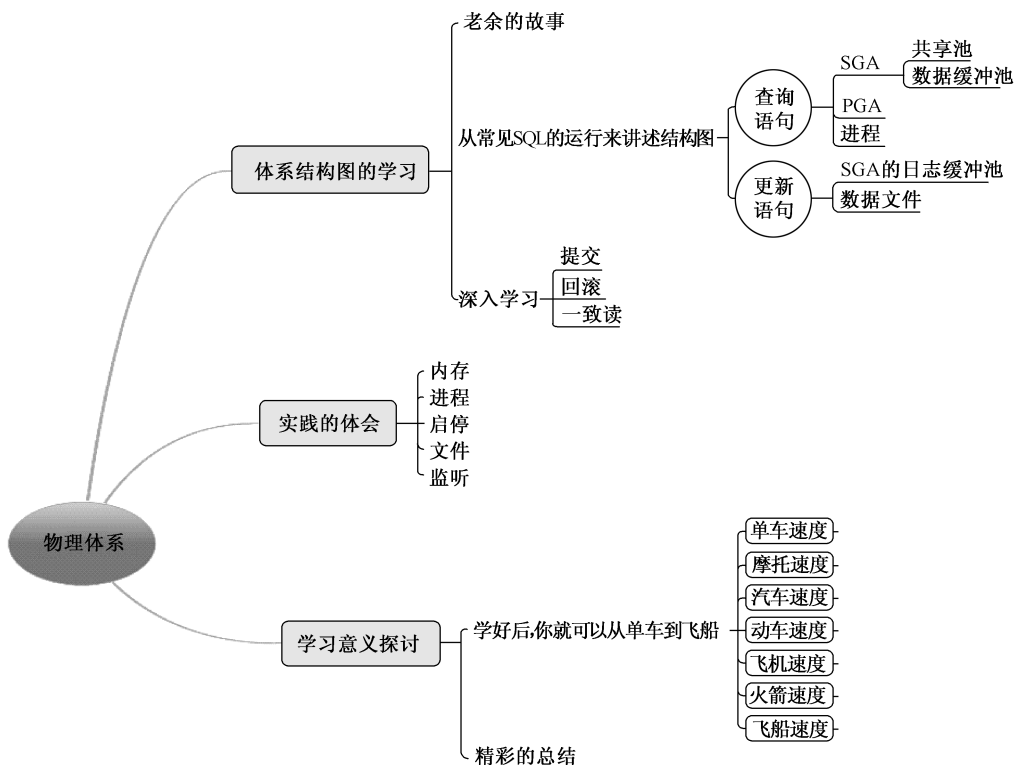


图 2-1 物理体系学习

“上了这么久的课，大家累不累啊？”梁老师笑着问大家。

“不累!”小莲脱口而出,她觉得这次培训还真有意思,时间也过得特别快。

“原来不累啊,那算了,如果累了就给大家说一个故事调节一下。”

“累啊!”

“好累啊!”

“累死了!”

“讲故事吧!”

这一小段时间的接触后,同学们都觉得这个梁老师很随和,很有趣,所以大家也真够大胆了,听说累了有故事听,台下来一片夹杂着阵阵欢笑的叫苦声,真是一个独特的风景。

“好吧,好吧,那讲故事吧。”梁老师笑了。

“我的故事肯定和 Oracle 课程是有关系的,否则要是随便乱讲,传出去了,梁老师岂不是要被开除了。

之前晶晶回答过我,说无论开发、管理、优化还是设计,基本原理都是必学的,回答得非常好。而基本原理中,体系结构又是首当其冲需要了解的知识,因此我的故事就和体系结构有关。在讲故事之前,首先让我们看看物理体系结构图是什么样子的,请看图 2-2。

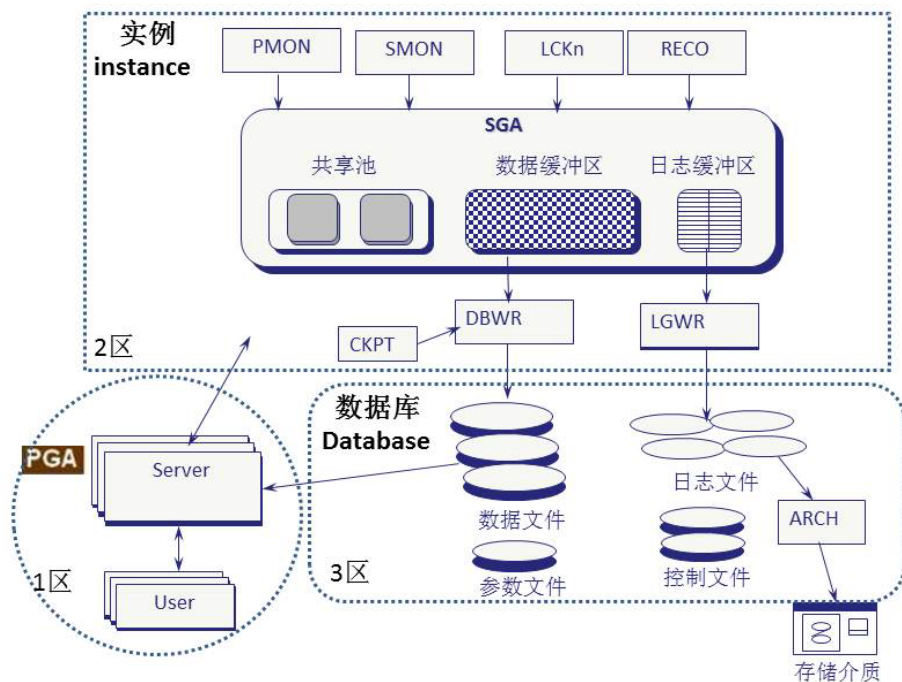


图 2-2 Oracle 体系结构图

这个体系结构图看似简单,其实很有玄机,我们平时遇到的数据库相关的各种问题,很多都

可以从体系结构中找到解决方法，理解体系结构非常重要。

这里我先简单描述一下 Oracle 的体系结构，我先粗略地说如下 5 点，大家要记得时刻回头看看这幅图。

- ① Oracle 由实例和数据库组成，我特意用两个虚框标记出来，上半部的直角方框为实例 instance，下半部的圆角方框为数据库 Database，大家可以看到我在虚线框左上角做的标注。
- ② 实例是由一个开辟的共享内存区 SGA（System Global Area）和一系列后台进程组成的，其中 SGA 最主要被划分为共享池（shared pool）、数据缓冲区（db cache）和日志缓冲区（log buffer）三类。后台进程包括图 2-2 中所示的 PMON、SMON、LCKn、RECO、CKPT、DBWR、LGWR、ARCH 等系列进程。
- ③ 数据库是由数据文件、参数文件、日志文件、控制文件、归档日志文件等系列文件组成的，其中归档日志最终可能会被转移到新的存储介质中，用于备份恢复使用。
- ④ 大家请注意看图 2-2 中的圆形虚线框标记部分的一个细节，PGA（Program Global Area）区，这也是一块开辟出来的内存区，和 SGA 最明显的差别在于，PGA 不是共享内存，是私有不共享的，S 理解为共享的首字母。用户对数据库发起的无论查询还是更新的任何操作，都是在 PGA 先预处理，然后接下来才进入实例区域，由 SGA 和系列后台进程共同完成用户发起的请求。

PGA 起到的具体作用，也就是前面说的预处理，是什么呢？主要有三点：第一，保存用户的连接信息，如会话属性、绑定变量等；第二，保存用户权限等重要信息，当用户进程与数据库建立会话时，系统会将这个用户的相关权限查询出来，然后保存在这个会话区内；第三，当发起的指令需要排序的时候，PGA（Program Global Area）正是这个排序区，如果在内存中可以放下排序的尺寸，就在内存 PGA 区内完成，如果放不下，超出的部分就在临时表空间中完成排序，也就是在磁盘中完成排序。

- ⑤ 我在图中标识了三块区域（大家注意看虚线框的左下角标注），分别是 1 区圆形虚线框，2 区直角方形虚线框，3 区圆角方形虚线框。用户的请求发起经历的顺序一般如下：
1 区→2 区→3 区；或者 1 区→2 区。”

“用户的请求为什么会有不访问 3 区 Database 区的情况啊？”小莲忽然想到，不禁脱口而出，打断了梁老师。

“这位同学提问得很好，不过我不告诉你答案。”

台下一片笑声……

“梁老师，为什么 SGA 内存区要分成共享池、数据缓冲区、日志缓冲区三部分啊，它们分别是什么作用？”胖小伙子敬昱忽地站起身来提问。

“这位同学提问得很好，不过……”梁老师故意放慢了语气。

“我不告诉你答案。”台下的同学异口同声地喊出来……

台下的又是一片笑声……

“安静安静，大家真聪明啊，都知道我后半段话要说啥。这两位同学提出来的问题，大家只要后续认真听讲，就自然有答案了。大家再看看这幅体系结构图，哦，不对，是大家闭上眼睛想想这幅体系结构图。”

台下同学全闭了眼，梁老师忍不住乐了，一群刚离开校园的新人，真是可爱。

“好了，大家先别闭了，看幻灯片，大家再多看两眼，回想一下我说的这 5 点，然后闭上眼睛回忆，再看看能否把这幅体系结构图牢记在心上，不能记全了再睁开眼睛看看，直到全部印在脑海为止。我们接下来描述的故事就和这个体系结构图紧密相关，记不住故事就不讲了。”

提及故事这两个字眼，台下异常安静，大家努力记忆这幅体系结构图，同时期待着梁老师的故事。

2.2 物理体系从老余开店慢慢铺开

2.2.1 老余的三个小故事

5 分钟过去了，“大家脑海中都记下这个体系物理结构图没？”梁老师微笑着问。

“记下了！”台下异口同声。

“好吧，让我给大家讲个故事吧。”

台下聚精会神。

2.2.1.1 顾客的尺寸

“话说有一个开成人服装店的老余，生意经营得还不错，他的店铺有一个特点，就是主要是熟客频繁光顾他的店铺，并且每次买的量还不少。这段时间放暑假，老余的儿子小余也来服装店帮忙，他的任务除了帮忙在门口招呼招呼客人外，还帮客人量腿长和身长，好给客人匹配合适尺寸的衣裤。

过了几天，小余问老余：‘老爸，我怎么老给客人量尺寸啊，他们自己都记不住吗，不会直接报给我吗？’老余一听，笑着说：‘我们这里的客人大多都不去记自己的尺寸，等你帮他们量好了，他们也不会去问具体尺码，就等着你提供他们合适尺寸的衣裤来试穿，都这样啊。’

‘老爸，那我可以把他们的尺码记录下来啊，下次他们来了，我只要一查就知道尺寸了，省了不少时间，还不会出错。’昨天早上小余没留神在量的过程中把尺寸给看错了，结果让客户试了一次很不合身才意识到，如果记录了客人某次合身试穿上衣裤的尺寸，下次直接用这个尺寸就可以避免了昨天早上的失误了，小余为此事耿耿于怀，因为昨天为此事没少给客人道歉。

‘不错啊，小家伙，我怎么没想到呢？’老余觉得是好主意，他想了想，又问，‘你准备记录在哪儿，本子上吗？’

‘那不行，本子里查询多不方便啊，说不定还不如我量一下快，我们可以录入在电脑甚至智能手机里啊。老爸，不知你听过没，有种软件叫数据库，录入查询都很方便啊。到时我们直接输入客户的名字，就能得到对应的尺寸信息，此外还可以把该尺寸和具体的衣裤的小、中、大、超大号对应起来，那就更方便了。’

‘成！就依你的主意，小家伙。’

在随后的一段日子里，不少熟客感觉到了变化，他们很少被要求量尺寸，经常在选择好款式后就直接拿到了衣裤开始试穿了，尺码每次都还非常匹配。这个变化让顾客更加满意了，因此店里的生意更好了。老余开心极了，不只是因为生意更好开心，发现儿子越来越能干才是真开心。

好了，故事说完了，好听吗？”梁老师笑眯眯地问大家。

“好听！”

“再来一个！”

台下同学们情绪非常放松，也无拘无束。

“还要再听啊，那好吧。”

“耶！”台下爆出欢呼声。

2.2.1.2 有效的调整

“话说老余的店铺规模还真是不小，各种衣裤是五花八门、琳琅满目，有时真让人挑花了眼，这也说明老余生意是真的不错，要不哪有资金搞这么大的规模呢？”

最近小余发现一个现象，店铺顾客是不少，生意也还不错！可是很多顾客逛了好长时间，才买到自己称心如意的服装，还有的就是转悠了好久，最后没买走人了。其实这种情况一直就是如此的，只是小余才刚留意到。

如果顾客能很快就挑到称心如意的服装，那对大家可真是皆大欢喜啊，小余开始观察并思考这个问题。

小余开始研究被买走的服装的类别，终于领悟到了什么叫潮流，近期被买得最多的款式最畅销！他明白他要怎么做了。

‘老爸，我有个想法。’

‘哦，又有啥好主意啊？乖儿子。’

‘晚上打烊时我们加加班，把最前排货架的服装全取下来，咱们整出空位子来，然后我要挑一些特定款式的服装摆到这最前排的货架上。’

‘儿子，为啥要这么整啊？’

‘老爸，你没注意到最近大家都买什么样款式的服装吗？我们要注意潮流，发现规律，把这

些被买得最多的系列款式放到最显眼的地方去，我们店铺这么大，选服装岂不是让人选得头晕眼花，其实现在很多人都是冲着这一系列款式来的，你没看出来吗？’

出于对儿子的信任，老余立即答应了。晚上他们忙得热火朝天，第二天，店铺有了新变化，店铺最前排的货柜被特别装修了一番，并且上面赫然写着‘店主推荐’四个大字。

在随后的一段日子里，老余惊喜地发现生意变得更好了。而小余也悄悄地做了统计，发现每位客人在店里逗留的时间也短了很多，大多都在店主推荐的货柜上快速找到自己喜欢的，就迅速离去了。不过小余和老余也没少做功课，他们不断地维护着这个店主推荐的货柜。陆续有不少服装从店主推荐货柜转到大厅里，也有不少从大厅中转入到店主推荐货柜里，他们的依据很简单，就是哪些款式买的人最多，哪些就留在店主推荐货柜里，很少买就转到大厅里。而大厅内也不是没有人买服装，他们观察大厅内的服装购买情况，哪个款式买的最多，很快就会被转到店主推荐货柜里。

后续还有新的改良，小余发现有一群顾客其实是冲着便宜来的，于是小余专门选了一个和店主推荐货柜差不多显眼的前排货柜，也特别装修了一番，写上‘超值优惠’四个大字。然后将成本比较低、卖价比较便宜的服装专门摆放在此处，这下方便了原来在大厅里到处寻觅相对便宜衣服的客户了，他们往往一进店门，就直奔超值优惠货柜，很快就心满意足地离去了……”

“好了，第二个故事也说完了，好听吗？”梁老师一脸微笑地问大家。

“好听！”

“有意思！”

“原来梁老师是教我们怎么做生意啊！”

“再来一个！”

台下七嘴八舌依旧无拘无束，不过他们听讲的时候，却是非常认真。

“还要再讲故事啊，你们怎么这么贪心！俗话说，事不过三，那我就再讲一个吧。”

“耶！”台下又爆出欢呼声。

2.2.1.3 记录的习惯

“老余的服装生意越做越红火，老余夫人余太太在学校附近新开的餐饮店却磕磕碰碰遇到些闹心的事，怎么说呢？”

原来这里不少学生由于平日里没支配好父母给的生活费，买了零食和玩具啥的，导致早餐及午餐时口袋里钱不够，于是不少孩子就赊账，有的时候记得第二天还了，有时第二天就没给补上，这样余太太就吃亏了，有时余太太明明记得孩子有欠钱，但是她也明白个中原因，也就算了。

最糟糕的是，居然有部分孩子由于经常赊账未兑现，最后弄得自己也不好意思，居然干脆不去余太太店里就餐，早上连中午一起饿肚子，这对孩子的成长影响多不好啊！

此外余太太发现另外一种情况，就是由于孩子胡乱开销父母给的费用后，孩子手头钱不够，

早餐和晚餐就开始减少食量，菜饭都减半，余太太看着这些还在长身体的孩子，急在心里。

你说余太太闹心不闹心，不仅是烦恼没赚钱，善良的她更烦恼这些孩子的健康问题。

小余听了妈妈的烦恼后，想到了后付费的方式，他让妈妈在店门口贴了一张告示，内容是允许孩子们就餐时暂缓付钱，到月底后，餐厅会让孩子将当月开销总费用明细带回家让父母确认，由父母通过网银或交给孩子带来支付给餐厅。

接下来，余太太还让就餐的孩子将事先准备好的协议单带回去给父母，里面首先描述了近期存在的孩子就餐中的不良情况，接下来描述了后付费的具体细节，最后还有填写联系方式及签字确认的相关内容。

情况的发展果然非常顺利，孩子吃完饭只需要在记有自己消费时间和金额的单子上签个字就可以离开了，之前的所有烦恼再也不存在了。孩子父母也更放心了，每月的结算也非常爽快。餐厅的顾客越来越多，越来越固定，生意越做越红火。

另外余太太餐厅的工作效率也大大提升了。为啥呢？原来余太太面对经常赊账的情况还要经常想法追讨，现在都不用操心这个了，等到月底，等着孩子的父母通过网银转账就好了。

余太太在这个餐厅的特色经营过程中，养成了记录的习惯，并且成了她生活中的习惯，认为有用的她都会尽量记录下来，这个习惯也帮了老余一个大忙。前段时间大暴雨导致城市内涝严重，老余赶紧手忙脚乱地雇人把店里所有服装转移出来，等灾害过后，服装要重新搬回店里时，如何摆设倒成了难题，原先可是有规划店主推荐货柜和超值优惠货柜哦，大厅的摆设也有一定的分布规律的。不过余太太在灾害来临尚未搬运期间对服装店的摆放做了详细的记录，还多角度拍了不少照片。这下老余小余父子俩就少了很多琢磨的时间了，在很短时间就部署摆设好了，又开始了正常的营业，没有耽搁太多的时间。”

“好了，第三个故事也说完了，故事到此结束了。”梁老师接着对大家说，“听完这三个故事，你们有啥感想？”

“要善于动脑筋。”

“要注意工作效率。”

“小余很聪明。”

“小余有生意头脑。”

.....

台下发言倒是非常踊跃，梁老师不回答，他在等着某个提问。

“梁老师，你说的三个故事和体系结构有啥关系啊？”

哈哈，终于等到了，梁老师笑了。

最后一个同学的发言倒是让大家都安静了下来，是啊，大家也都开始思考这个问题，三个故事和体系结构有啥关系，是如何结合在一起的呢？

2.2.2 体系结构原理初探

2.2.2.1 从一普通查询 SQL 说起

“同学们，关于故事与体系物理结构的联系我们暂且先放一放，现在我们开始描述 Oracle 的体系结构，你们现在有没有忘记刚才说的体系结构图啊？”

“记得！”对于这些年轻的新人来说，记性好是他们最大的优势之一，5 分钟时间强记这个看上去不是非常复杂的结构图，大多数人都做得到。

“很好，这里我先问大家一个问题，大家在校园和最近实习过程中，有接触过 SQL 语句的同学请举手。”

台下基本上所有的同学都举手了。

“我发现 Oracle 数据库有不少人没接触，但是标准 SQL 倒是大多数人都接触过了。”梁老师笑着说道。

其实梁老师心里明白这是什么原因，这是因为大学里很少会开设具体的数据库课程（如 Oracle、DB2、SQL Server、Informix），即便 Oracle 占据了 70% 以上市场份额也不例外。标准 SQL 由于较为通用，所以大学一般会开设这样的课程，此外数据库基础这样的课程（不和具体数据库挂钩的），大学也会开设。因此很多人没接触过 Oracle，大多数人接触过 SQL 这一现象并不奇怪。

“很好！大家都有一定的 SQL 基础知识，我就以一些 SQL 发起的始末，来和大家谈谈 Oracle 体系结构。”

“大家先看一条最简单的 SQL 查询语句 `select object_name from t where object_id=29;`，当你发出一条上述 SQL 指令后，该 SQL 语句从 1 区先做准备工作，如图 2-3 所示。

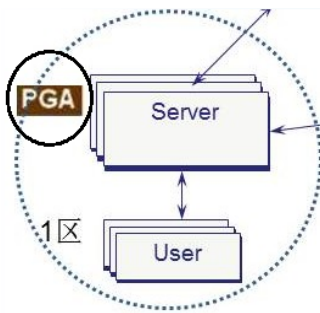


图 2-3 PGA 区说明

前面我向大家描述过 PGA 不同于 SGA，是仅供当前发起用户使用的私有内存空间，这个区域的具体作用有三点，忘记的同学打一下屁股，然后回头看看课件记录。这里该连接只是完成了两点，即用户连接信息的保存和权限的保存，别小看了这个保存的重要性，只要该 SESSION 不断开连接，下次系统不用再去硬盘中读取数据，而直接从 PGA 内存区中获取。

此外该 SQL 还会立即匹配成一条唯一的 HASH 值，接下来该 SQL 指令进入 2 区进行处理，首先敲开 SGA 区的共享池的大门，准备登门拜访，如图 2-4 所示。



图 2-4 共享池说明

共享池的大门打开了，该 SQL 先在房内查询是否什么地方有存储过这个 SQL 指令的身份证（就是那个唯一的 HASH 值），如果没有，那就要辛苦了，首先查询自己的语句语法是否正确（比如 from 是否写成了 form）、语义是否正确（比如 id 字段根本就不存在）、是否有权限，在这些都没问题的情况下生成这条语句的身份证，唯一的 HASH 值就被存储下来了。接下来开始进行解析，解析什么呢？比如 select object_name from t where object_id=29 这个语句，在 object_id 列有索引的情况下，是用索引读更高效呢，还是全表扫描更高效？Oracle 要做出选择。

Oracle 处理这个事情的依据很简单，就是把两种方式都估算一遍，看哪个代价（COST）更低，就用哪种。好比上街买菜，有的菜摊白菜卖 5 毛/斤，有的卖 6 毛/斤，那就选 5 毛的。不过这里可不是真正地分别执行 2 次来比较，具体方法在后续涉及优化的学习中大家会了解到。

假设 Oracle 认定使用索引代价（COST）更低，只要 5 毛钱，于是 Oracle 就选用了索引读的执行计划而放弃了需要 6 毛钱的全表扫描方式。接下来这个索引读的执行计划就立即被存储起来，并且和之前存储的该 SQL 的身份证（唯一 HASH 值）对应在一起。

接下来，该 SQL 指令好比钦差大臣一样，手持‘索引读获取某某数据’这个圣旨，继续往前走，他这是要去哪呢？原来是直奔‘数据缓冲区’府内去宣读圣旨了，如图 2-5 所示。

数据缓冲区开门迎接跪谢天恩后，立即要根据 ID 列上的索引从 t 表中查找 object_id 值为 29 的宝物，但是所要的东西府内找不到，怎么办呢？数据缓冲区府只好传令下去，八百里加急赶去偏远的 Database 军营的数据文件区去查找皇上要的东西（当然，必须用索引读的方式查找，这是圣旨，不可违抗）。如果查到了，就带回数据缓冲区府，并由钦差大臣展现给皇上，如果找不到，也只有就此复命，如图 2-6 所示。

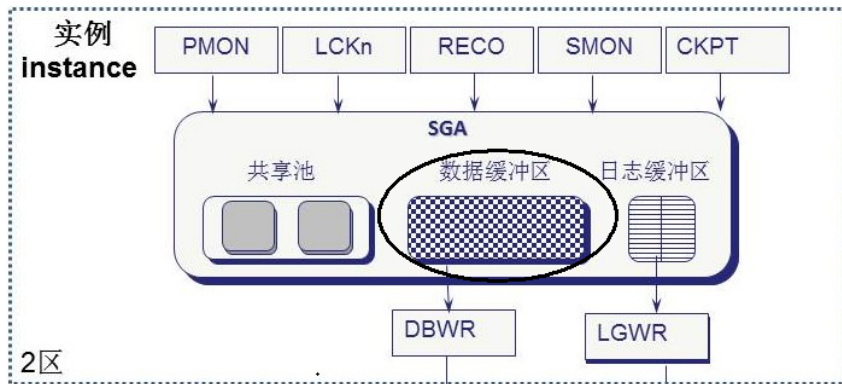


图 2-5 数据缓冲区说明

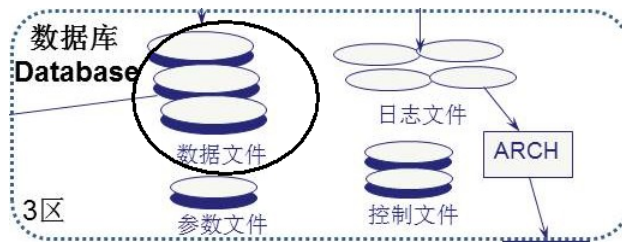


图 2-6 数据文件查数据

至此，一条最普通的 SQL 指令的前后经历就说完了，大家听得明白吗？有什么疑问吗？”梁老师的面容始终充满微笑。

“听明白了！”

“梁老师，我还是没体会到您之前说的故事和您描述的体系结构有啥关系？”小莲同学前面的课程记得真牢，有些不依不饶啊。

“梁老师，您说 Oracle 会选择代价 COST 最低的一种执行计划，方法是分别尝试一下，比较出高低，但是又不是真正执行，这是咋做到的啊，会准确吗？”

“梁老师，我听是听明白了，不过怎么体系结构图的 2 区 SGA 区的日志缓冲区没说到啊？”

“哦，还有还有，3 区 Database 区怎么就说描述一个数据文件啊，其他组件呢？”

“梁老师，您说实例是由一个内存区 SGA 和一组进程组成的，您刚才怎么也都没说进程啊，你看 DBWR、LGWR 等，都没说啊？”

“梁老师，这个语句也太简单了吧，复杂的语句也是如此吗？”

.....

“很好！老师发现大家上课非常认真，前面的体系结构图也记得非常牢，思维也非常活跃，老师非常满意！”梁老师开心地说道，“不过我确实是把一条最常见的 SQL 指令在数据库中

的前后经历描述给大家了，期间并没有描述错，大家却有这么多疑问，你们觉得问题出在哪里呢？”

台下安静下来，大家都在思考。

2.2.2.2 老余故事终现用心良苦

“疑问总会得到解决的，刚才某个用户发出那条 SQL 指令的执行描述大家都听明白了吧，之前和大家海阔天空地聊了不少，现在我们就具体做一些试验，让大家有一个直观的感受。

下面我就来当那个用户，由我来执行这个 ‘select object_name from t where object_id=29’ 的 SQL 指令。我构造的试验环境如下，所有的人都可以根据我的准备工作完成这个试验，请同学们回去后务必自己动手操作一遍：

```
sqlplus ljb/ljb
drop table t ;
create table t as select * from all_objects;
create index idx_object_id on t(object_id);
set autotrace on
set linesize 1000
set timing on
select object_name from t where object_id=29;
```

这里我简单介绍一下上述脚本，首先是登录数据库，sqlplus ljb/ljb 就是我的连接，接下来是建表构造数据和建索引的过程，随后的 set autotrace on 是开始跟踪 SQL 的执行计划和执行的统计信息，而 set timing on 是表示跟踪该语句执行完成的时间。这些都准备好了，最后关键语句 select object_name from t where object_id=29 就可以粉墨登场了。

这些构造脚本的试验方法这里先只是简单提一下，将来的系列培训还会提到的，准备试验脚本也是我多年的习惯，要善于构造出试验，来研究分析事物本质，此外大家尽量在自己的笔记本环境中安装一下数据库软件，Oracle 10g 和 Oracle 11g 都可以，这样才方便自己随意做试验而不影响他人环境。

好了，话不多说，现在我们开始执行脚本，让大家直观地体会一下感受，试验分析过后，大家的不少疑问都会自然而然地消除了，大家期待吗？”

“期待！” 台下异口同声。

“好，那我们开始吧，我们就简单地先在 Windows 平台做操作吧，不管是 Windows 还是 UNIX 平台，数据库的原理都是一样的，另外大家别忘记了在自己的笔记本上安装 Oracle 数据库，然后执行我课上的脚本，来实践我上课的内容。

我们进入 cmd 命令行，复制粘贴上述脚本，执行结果如下：

```
C:\Users\ljb>sqlplus ljb/ljb
```


收获，不止 Oracle

```
SQL> drop table t ;
表已删除。
SQL> create table t as select * from all_objects;
表已创建。
SQL> create index idx_object_id on t(object_id);
索引已创建。
SQL> set autotrace on
SQL> set linesize 1000
SQL> set timing on
SQL> select object_name from t where object_id=29;
OBJECT_NAME
-----
C_COBJ#
已用时间: 00: 00: 00.04
执行计划
-----
Plan hash value: 2041828949
-----
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-----
| 0 | SELECT STATEMENT | | 1 | 30 | 2 (0)| 00:00:01 |
| 1 | TABLE ACCESS BY INDEX ROWID | T | 1 | 30 | 2 (0)| 00:00:01 |
| * 2 | INDEX RANGE SCAN | IDX_OBJECT_ID | 1 | | 1 (0)| 00:00:01 |
-----
Predicate Information (identified by operation id):
-----
2 - access("OBJECT_ID"=29)
Note
-----
- dynamic sampling used for this statement
统计信息
-----
52 recursive calls
0 db block gets
73 consistent gets
4 physical reads
0 redo size
420 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 2-1 SQL 语句首次查询的情况

接下来我们再执行一次 select object_name from t where object_id=29 的 SQL 指令。

```
SQL> select object_name from t where object_id=29;
```

```
OBJECT_NAME
```

```
C_COBJ#
```

```
已用时间: 00: 00: 00.01
```

```
执行计划
```

```
Plan hash value: 2041828949
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	30	2 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	T	1	30	2 (0)	00:00:01
* 2	INDEX RANGE SCAN	IDX_OBJECT_ID	1		1 (0)	00:00:01

```
Predicate Information (identified by operation id):
```

```
2 - access("OBJECT_ID"=29)
```

```
Note
```

```
- dynamic sampling used for this statement
```

```
统计信息
```

```

0 recursive calls
0 db block gets
4 consistent gets
0 physical reads
0 redo size
420 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 2-2 同一 SQL 再次查询后性能提升

好了，该 SQL 指令前后分别被执行 2 次了，大家认真观察前后 2 次输出的结果，找找看有没有什么差异？记住，这可是一模一样的语句，返回一模一样的结果哦。”

“执行时间第一次更长！”

“统计信息方面两次执行好像差别很多……”

“非常好，大家观察得很仔细！”梁老师满意地说道。

“大家基本上都说到了，下面看我分析贴出比较图，左边显示 0.04 秒为第一次执行时间，而右边 0.01 秒为第二次完成时间，比第一次快了整 3 倍（如图 2-7 所示）。

SQL> select object_name from t where object_id=29; OBJECT_NAME ----- C_COBJ# 已用时间: 00:00:00.04	SQL> select object_name from t where object_id=29; OBJECT_NAME ----- C_COBJ# 已用时间: 00:00:00.01
--	--

图 2-7 比较执行时长

接下来看统计信息的比较，首次执行产生了 52 次递归调用、73 次逻辑读、4 次物理读。而第 2 次执行递归调用为 0，逻辑读为 4 次，物理读为 0（如图 2-8 所示）。

- dynamic sampling used for this statement 统计信息 ----- 52 recursive calls 0 db block gets 73 consistent gets 4 physical reads 0 redo size	- dynamic sampling used for this statement 统计信息 ----- 0 recursive calls 0 db block gets 4 consistent gets 0 physical reads 0 redo size
---	---

图 2-8 比较递归调用与逻辑读

看来前后两次执行差异非常大啊，同学们，这是为啥呢，真正原因在于第 2 次执行实现了同样的目的却少做了不少事，少做了哪些事呢？大家脑海里要再次浮现体系结构图，否则跟不上我讲课的思路哦。

SQL 指令的执行我之前已经描述了，现在我是专门描述第 2 次执行的差异。

- ① 用户首次执行该 SQL 指令时，该指令从磁盘中获取用户连接信息和相关权限信息权限，并保存在 PGA 内存里。当用户再次执行该指令时，由于 SESSION 之前未被断开重连，连接信息和相关权限信息就可以在 PGA 内存中直接获取，避免了物理读。
- ② 首次执行该 SQL 指令结束后，SGA 内存区的共享池里已经保存了该 SQL 唯一指令 HASH 值，并保留了语法语意检查及执行计划等相关解析动作的劳动成果，当再次执行该 SQL 时，由于该 SQL 指令的 HASH 值和共享池里保存的相匹配了，所以之前的硬解析动作就无须再做，不仅跳过了相关语法语意检查，对于该选取哪种执行计划也无须考虑，直接拿来主义就好了。

- ③ 首次执行该 SQL 指令时，数据一般不在 SGA 的数据缓存区里（除非被别的 SQL 读入内存了），只能从磁盘中获取，不可避免地产生了物理读，但是由于获取后会保存在数据缓冲区里，再次执行就直接从数据缓冲区里获取了，完全避免了物理读，大家可以注意到我们的试验，首次执行物理读为 4，第 2 次执行的物理读居然为 0，没有物理读，数据全在缓存中，效率当然高得多！

所以，这就是前后两次试验执行效率差异如此之大的原因，原来是和 Oracle 的体系结构有密切关系，原来 Oracle 的体系结构设计是有玄机的，这种设计保证了再次执行的效率能大大提升，是非常有意义的。

另外大家不是有很多疑问吗，现在可以问大家一下，之前我说的故事《顾客的尺寸》、《有效的调整》的思考方式，和刚才描述的体系结构相通吗？请大家思考。”

“梁老师，《顾客的尺寸》讲述了小余首次丈量顾客尺寸时将尺寸记录下来，等下次遇到同一顾客时就直接快速查出该顾客尺寸，从而无须每次重复做这个丈量的动作的故事，这不是和共享池一模一样嘛，缓存避免了再次解析这个耗时动作。”小莲恍然大悟。

“梁老师，我也来说说。数据缓冲区获取数据比磁盘获取数据要快好多，越经常获取的数据往往意味着是越有意义的数据，后续再次查询这些数据的概率比较大，而《有效的调整》中小余把顾客经常购买的服装款式放到最显眼、方便获取的大厅前排货柜里。在整个大厅寻找和在前排货柜寻找的差异类似于数据缓冲区查找数据和磁盘查找数据的差异！”晶晶也如梦初醒了，她也明白了梁老师讲这个故事的用心。

“看来我的故事没白讲啊。”梁老师这句话显然是肯定了两个同学的回答，台下同学们也纷纷点头，表示赞许。

2.2.2.3 一起体会 Oracle 代价

“在之前介绍体系物理结构时有不少同学提问，有问体系结构和我说故事有啥关系的，不过这个大家自己搞明白了。还有问 Oracle 判断哪种执行计划更高效时，是用代价来判断的，问基于什么原理，是否准确。这个问题我想在这里和大家一起研究一下。当然还有不少同学有其他疑问，我们留在后续再分析研究。

判断代价高低基于什么原理我们先暂缓讨论，首先测试一下 Oracle 对代价 COST 的判断是否准确，这样多少让我们对 Oracle 数据库有些信心。

我们知道在表有索引的情况下，Oracle 可以选择索引读，也可以选择全表扫描，这是两种截然不同的执行计划，不见得一定是索引读胜过全表扫，有时索引读的效率会比全表扫更低，所以 Oracle 的选择不是看是啥执行计划，而是判断谁的代价更低。

让我们设计一组试验来证实一下 Oracle 的代价判断是否准确。前面我做组一组试验，证实了执行某 SQL 指令后，再继续执行同样的 SQL 指令，性能会大幅度提升。

现在我还是继续这个例子，同样的 T 表，同样的 SQL 语句，但是我要加上一个 HINT（什么

收获，不止 Oracle

叫 HINT，就是一种强制写法，比如强行让某 SQL 语句不走索引，或者强行让某 SQL 语句走某索引)。原来的 SQL 语句如下：

```
select object_name from t where object_id=29;
```

现在我增加/*+full(t)*/的写法，来强制该 SQL 语句不走索引，走全表扫描，具体如下：

```
select /*+full(t)*/ object_name from t where object_id=29;
```

执行这个语句的目的是看看，Oracle 选择索引是否是明智正确的选择，是否会更快，逻辑读等是否都少得多。

准备脚本如下，请注意我故意让 select /*+full(t)*/ object_name from t where object_id=29;执行两遍，是为了多次执行后消除共享池的解析、减少甚至消除物理读以及递归调用：

```
sqlplus ljb/ljb
set autotrace on
set linesize 1000
set timing on
select /*+full(t)*/ object_name from t where object_id=29;
---以下是故意再执行一遍
select /*+full(t)*/ object_name from t where object_id=29;
```

脚本 2-3 故意强制走全表扫描的情况

我们进入 cmd 命令行，复制粘贴上述脚本，执行结果如下，我只截取了第 2 次 SQL 指令执行后的部分信息：

```
SQL> select /*+full(t)*/ object_name from t where object_id=29;
OBJECT_NAME
```

```
-----
C_COBJ#
已用时间: 00: 00: 00.12
执行计划
```

```
-----
Plan hash value: 1601196873
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	
0	SELECT STATEMENT		1	30	174 (1)	00:00:12	
* 1	TABLE ACCESS FULL	T	1	30	174 (1)	00:00:12	

```
-----
Predicate Information (identified by operation id):
```

```
-----
1 - filter("OBJECT_ID">=29)
```

Note

 - dynamic sampling used for this statement
 统计信息

```

0 recursive calls
0 db block gets
765 consistent gets
0 physical reads
0 redo size
420 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
  
```

图 2-9 至图 2-11 的左边均为 HINT 强制走全表扫的 SQL 指令，右边均为此前 Oracle 自己选择用索引方式的 SQL 指令。

SQL> select /*+full(t)*/ object_name from t where object_id=29;	SQL> select object_name from t where object_id=29;
OBJECT_NAME	OBJECT_NAME
-----	-----
C_COBJ#	C_COBJ#
已用时间: 00:00:00.12	已用时间: 00:00:00.01

图 2-9 强制 FULL SCAN 比较时长

比较强制走全表扫的 SQL 指令和此前 Oracle 自己选择用索引方式的 SQL 指令的执行时间，以及各自执行代价及逻辑读的大小，发现强制走全表扫描明显在性能上要大大逊色于 Oracle 自己选择的索引方式。

其中最为重要的是图 2-10，左边的代价为 **174**，右边的代价为 **2**，显而易见左边的代价高得多，所以 Oracle 当然会抛弃使用全表扫描的方式来检索数据，转而使用代价小得多的索引读。

执行计划

Plan hash value: 1601196873

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	30	174	(1) 00:00:12
* 1	TABLE ACCESS FULL	T	1	30	174	(1) 00:00:12

执行计划

Plan hash value: 2041828949

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	30	2	(0) 00:00:01
1	TABLE ACCESS BY INDEX ROWID	T	1	30	2	(0) 00:00:01
* 2	INDEX RANGE SCAN	IDX_OBJECT_ID	1		1	(0) 00:00:01

图 2-10 比较代价

由此我们得出结论，Oracle 对代价的评估和判断是相当准确的：代价 174 的情况下，执行时间需要 0.12 秒，远高于代价 2 的情况下的 0.01 秒；代价 174 的情况下，逻辑读为 765，也远高于代价为 2 情况下的逻辑读 4（如图 2-11 所示）。

- dynamic sampling used for this statement	- dynamic sampling used for this statement
统计信息	统计信息
0 recursive calls	0 recursive calls
0 db block gets	0 db block gets
765 consistent gets	4 consistent gets
0 physical reads	0 physical reads

图 2-11 比较逻辑读

这个小试验证明了 Oracle 数据库还是可以信赖的，我们可以充分相信它的选择，只是它的选择比较艰难，开销很大，如果它们选择的结果能保留下来重用，那艰难的动作就可以避免重复操作，性能就能大幅度提升了，Oracle 在这方面的设计确实很巧妙。

同学们，有谁听不明白的吗？”梁老师结束了这个 COST 关键概念的介绍，笑着问大家。

“没有！”场下异口同声。

确实，场下的小伙子小姑娘年轻好学、理解力强，梁老师的描述又如此通俗易懂、风格又如此轻松愉快不会让人乏味和走神，谁还不能接受呢？

2.2.3 体系结构原理再探

2.2.3.1 从一普通更新语句说起

“关于体系结构，我已经用很通俗的方式给大家粗略地描述了一遍，从同学们的反应交互来判断，我所描述的大家应该都听懂了，大家确实都非常聪明！”

不过在前面描述体系结构时，大家有关于体系结构图的三个疑惑，老师还没给大家答疑：SGA 区的日志缓冲区没描述；系列后台进程的功能都没提及；Database 区数据库文件之外的其他组件也没被提及。

其实这三条疑问我可以汇总成一条，即：梁老师的描述中，体系结构图中不少组件根本没被提及，难道这些组件是多余的吗？

请问之前提问的三个同学，我这么汇总对吗？”

“对！”三个提问的同学齐声应到。

“你们知道为什么我当时不回答你们这些疑问吗？那是因为你们的疑问其实是由于梁老师的首次描述太粗略了，如果梁老师的描述再细致一点，你们就没疑问了。

我的风格是循序渐进，先说最简单的、最重要的、最易理解的部分，让大家先理解一部分，

然后再略为深入、接下来再深入，直至最后全面深刻理解完毕。而且这个循序渐进的过程我希望伴随着大家积极的思考、提问，我认为这样课程的效果是最好的！”

听到这里，小莲一个劲点头。她感觉这次培训的生动活泼、引人入胜程度，是她之前任何一次培训所没经历过的，而且她也喜欢上了数据库的学习和研究，觉得其乐无穷。

“select object_name from t where object_id=29 是我刚才举的例子，该 SQL 语句还具体执行了一下，让大家直观地感受了一下，不过不知道大家有没有注意到，这个 SQL 语句仅仅是一条查询语句，你们觉得除了查询语句，还应该有啥语句？”

“更新语句！”台下回答很响亮。

“是啊，SQL 语句中除了查询语句，还应该有更更新语句，否则这个数据库就要变成一个只读数据库了，如果是一个只读库，那组件当然不需要那么多了，这就可以解释你们的疑问了。

此外更新又可分为插入、修改、删除三类，这些动作都属于数据库操作中最常见的操作，现在我准备举一个更新语句的例子，来继续描述体系结构。下面我们仅以修改语句 update t set object_id=92 where object_id=29;为例。

回想之前的 select object_name from t where object_id=29，大家应该可以很清楚地想象到这个 update 语句的前后经历：

- ① 如果该用户并没有退出原连接去新建立一个连接，PGA 区的用户连接信息和权限判断等诸多动作依然不用做，否则需要完成用户连接信息和权限判断等诸多动作。
- ② 如果该语句是第一次执行，在共享池里依然需要完成语法语意分析及解析，update t set object_id=92 where object_id=29 指令中想匹配到 object_id=29 的记录既可以用索引读，也可以用全表扫描，到底选用哪种执行计划需要根据代价的大小来选择。
- ③ 接下来进入数据缓冲区，首次执行该数据一定不在缓冲区里，也是和前面一样，先从磁盘中获取到缓冲区中……

以上三点和之前的 select object_name from t where object_id=29 描述几乎没有任何本质区别，差异在于查询语句做完这三步后，返回数据结果给用户，就收工回家休息了。而更新语句的工作却远没结束，还要流着汗水继续工作。

接下来，更新语句要完成什么动作呢？

大家看图 2-12 的方框部分，终于涉及之前未描述的部分了，DBWR 进程，看来系统后台进程未被提及的历史终于被改写了！

其实描述起来特别简单，就是在数据缓冲区内修改完数据后，会启用 DBWR 进程，完成更新的数据从内存中刷入到磁盘，将磁盘中的 object_id=29 的值更新为 92。因为磁盘才是真正存储数据的地方，否则一断电，数据在内存中，那就灰飞烟灭了。

接下来请大家继续看图 2-13 中我标记出的方框部分，日志缓冲区和 LGWR、ARCH 后台进程及日志文件都出现在议题上了，人物情节越来越丰富，之前未被提及的人物尽数粉墨登场了。

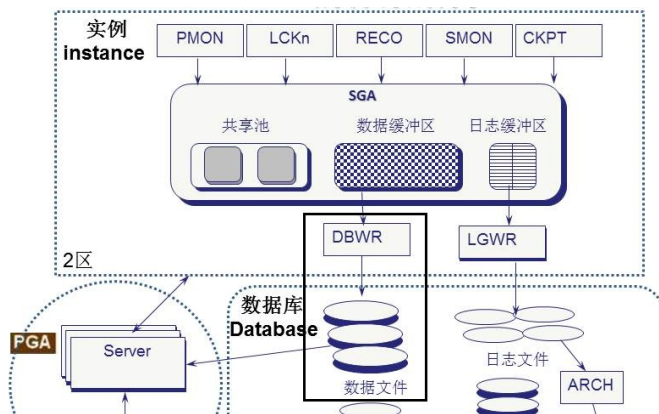


图 2-12 DBWR 写

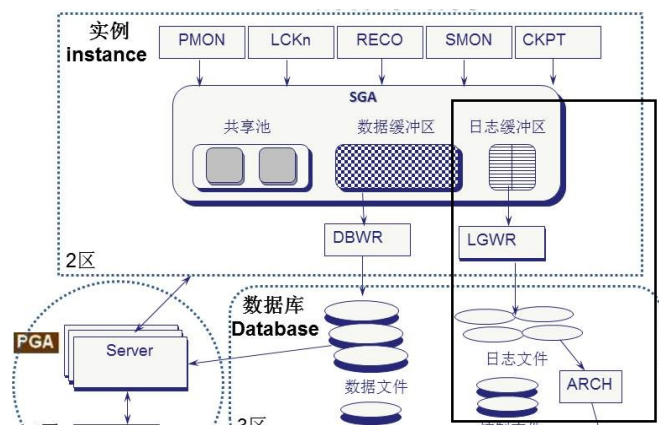


图 2-13 写日志

日志缓冲区保存了数据库相关操作的日志，记录了这个动作，然后由 LGWR 后台进程将其从日志缓冲区这个内存区写进磁盘的日志文件里。目的很简单，就是为了便于将来出现异常情况时，可以根据日志文件中记录的动作，再继续执行一遍，从而保护数据的安全。

举个简单的例子，新建好一个数据库，完成如下三个动作：

- ① A 动作，建立一张表 T；
- ② B 动作，插入一条数据进 T 表；
- ③ 将该数据更新某字段。

A、B、C 三个动作都被记录到日志中。接下来数据库出现异常，比如 T 表记录被人误删除了，怎么办？

很简单，就是根据日志把 B、C 再执行一遍就好了。

那如果 T 表整张表都被人给删除了，怎么办？

也很简单，就是根据日志把 A、B、C 三个动作再执行一遍就好了。

由此可见，Oracle 写日志是很重要的，LGWR 进程是联系日志缓冲区和日志文件的辛勤工作者，请看图 2-14，我把日志文件标记上 1、2、3、4 四个标记，如下：

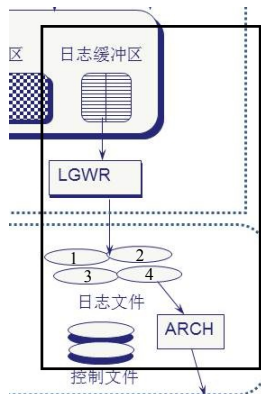


图 2-14 归档切换

一开始我们写日志文件 1，写满切换到日志文件 2 继续写，2 写满后写 3，3 写满后写 4。当 4 也写满后，该怎么办呢？”梁老师到这里突然停下来问大家。

“4 写满后再写 1 啊！”马上有人抢答。

“那 1 的数据呢，被覆盖？”梁老师问。

“会不会是先把 1 的数据备份出去，再覆盖重写？”有同学不敢确定地小声回答。

“完全正确，这个 ARCH 就是 1 在被重写时先备份出去的文件，命名为归档文件，接下来再到 2、3、4 我们就可以依此类推了。此外这些 ARCH 文件也需要定时转移到新的存储介质，这个存储介质里的 ARCH 就是我们将来数据库故障恢复的法宝了。听明白了吗？”

“听明白了！”大家回答得很响亮。

“那你们将眼睛闭起来，回忆一下体系结构图，脑子里分别回想我举过的查询语句和更新语句的例子，体会一下是否感觉亲切多了，也更容易记忆了。”梁老师让大家做一个尝试。

原来数据库的主要组件的功能是这样的，小莲瞬间豁然开朗了！她闭上眼睛，脑子像放电影一样过了一遍查询语句和更新语句在数据库的执行过程，顿时觉得这个体系结构图一下子就从脑海里展现出来，清晰极了。

2.2.3.2 体系结构中提交的探讨

“大家都理解 Oracle 体系物理结构的原理了吗？”梁老师问。

“理解了！”同学们回答得倒是很自信。

“其实大家只是了解了一个大致原理，很多细节还不知道，我还以刚才那个更新语句 `update t set object_id=92 where object_id=29` 为例，如果我执行完这条指令，你们可以在数据库中查询到 `object_id=29` 被改变为 92 了吗？”

“可以！”大部分同学都这么回答。

“不可以，因为没提交。”少数几个同学持反对意见。

“那我自己查数据库，可以查到 `object_id=29` 被改变为 92 了吗？”梁老师继续问。

“也不可以！”

“哦，如果还是发起更新语句的本 SESSION 做的查询，这倒是可以查到变化的。”梁老师笑着说道，“这个大家后续简单地做个试验就可以证明了。”

“其实这里的机制涉及的细节非常多，说得太细会和我今天上课的预想冲突，后续会有很多机会和大家细细探讨的。当然也不是所有的细节都需要了解的，了解实用的、有借鉴意义的细节即可。我还是以我的风格来授课，在引导思索中让大家慢慢地越了解越多，最后贯穿起来灵活应用。

一条更新语句无论插入、修改还是删除，最终执行完毕后都需要执行用户做提交 `COMMIT` 或回滚 `ROLLBACK` 的确认，前者表示用户确认无误了，确实需要更新。后者表示用户后悔了，赶紧撤销刚才的动作。

回滚的场景我后面再描述，先说提交，在此我先问大家一个问题，当你发起提交命令后，你觉得数据缓存区的数据一定会立即被 `DBWR` 进程写进磁盘吗（如图 2-15 所示）？

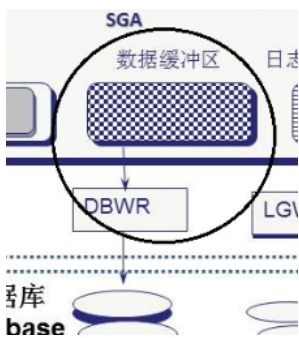


图 2-15 数据缓冲区写出机制

台下有的说会，有的说不会，七嘴八舌，大家都等着梁老师确定答案。

“那我举个例子吧，比如我去银行存自己辛苦赚的 10 万元血汗钱（之前户头已经有 10 万元），

柜台工作人员在你签字确认后，收走了你 10 万元钱，你就空手回去了。

而工作人员做的动作无非就是执行一条 10 万元记录的插入语句或是执行一条从原先的 10 万变为 20 万的 UPDATE 语句。由于你签字确认了，工作人员执行了 COMMIT 的动作；如果你在签字确认前忽然提出不想存了，工作人员无非就是执行一条 ROLLBACK 语句，10 万元钱你带回去就好了。

刚才有人说 COMMIT 后数据不一定会立即被写进磁盘，换句话说，就是还保留在 SGA 的数据缓存区里。

那接下来我的故事就有些不幸了，当我离开银行时银行忽然断电，可是我的新存 10 万元存款的记录还在数据缓存区这个内存区域里，断电的后果是内存里存储的任何信息都消失了。

我回家的路上心里很不踏实，10 万可是我的血汗钱啊，于是去某个 ATM 机再去查询看看总金额是否是 20 万，结果查询后，还是 10 万，于是愤怒地返回银行。”

说到这里，梁老师停了下来，问大家：“现在你们觉得提交后数据会立即从数据缓存区写到磁盘吗？”

“会！”这下大家回答得相当一致。

“为什么？”

“因为提交表示最终确认，此时数据依然存放内存中断电会丢数据，放磁盘中保险。”小莲回答问题倒是非常快速。

“那我宣布答案。”梁老师笑了笑说道，“所有人都错！”

“不是吧，居然是不会刷入磁盘，那断电了咋办？刚才梁老师你不是提示我们肯定是会刷进磁盘吗？”

“梁老师，我觉得肯定会刷入磁盘，你不要你的 10 万元血汗钱了啊？”

台下笑声一片。

“原来是心疼梁老师的血汗钱啊，很感动。”梁老师笑着调侃道，“不过说不会刷入磁盘的那位同学回答也是错的，答案是也许会也许不会，不一定。这里我问大家一个问题，数据缓存区的数据每提交一次就刷出一次和积累到一定量后成批刷出，哪个性能更高？”

“当然是批量做效率更高！”同学们回答得很干脆。

“回答正确！大家还记得之前梁老师说过的《记录的习惯》故事吧，余太太可以每天都拿着孩子们的账单去找家长兑现钱，也可以选择每月甚至每季度去一次，前者会把余太太累晕，后者则轻松得多了，效率孰高孰低自是不言而喻。

所以答案是 COMMIT 无法左右数据何时从数据缓存区刷入到数据区，Oracle 根据一定的规则来促成这个动作，就是缓存区的数据积累到一定的程度，再批量刷入到磁盘中，因为这样高效得多，大家听明白了吗？”

“听明白了，只是难道 Oracle 数据库不怕断电吗？内存中的数据最怕断电啊梁老师。”小莲脱

收获，不止 Oracle

口而出。

“很好，同学们担心的是安全问题，安全和效率很多时候是不能兼顾的，在无法兼得的情况下，我们肯定是牺牲效率换取安全，可是 Oracle 在这里做到了兼顾，所以才可以在批量刷出而不怕断电危机，大家想想看 Oracle 是如何做到的。”

台下一片寂静，暂时没人回应。

“梁老师提醒大家一下，还记得梁老师说的第三个故事《记录的习惯》吧，余太太是每天拿记录本找孩子家长结算效率高还是每个月底或者季度清算一次效率更高？”

“当然是每个月底或季度清算一次效率更高！”同学们回答得毫不含糊。

“那余太太为什么放心每个月底或者是季度和孩子家长结算一次呢？”梁老师继续提问。

“梁老师，因为她记录下来了，有明细单及孩子的签字，之前还与家长签订了合同，家长无从抵赖，所以可以放心啊。”曾祥毫不迟疑地举手回答了。

“回答得非常好，那 Oracle 数据库呢？”梁老师继续引导着。

“梁老师，您之前说过 Oracle 有日志缓存区和日志文件，这不就是余太太的记事本吗？只是日志缓存区也是内存区，也怕断电，但是日志文件是永久保存在物理磁盘中的，不怕断电。

因此我认为 COMMIT 时日志缓冲区肯定会把要操作的动作写到磁盘的日志文件里，这样 Oracle 就不一定非要将数据从数据缓存区写到磁盘了。磁盘的日志文件不是内存中的日志缓冲区，是永久保存不怕断电的，断电后可以依据磁盘里的日志文件重新操作一次，把刚才数据缓存区丢失的数据恢复。

所以数据缓冲区是可以批量刷出的，效率和安全可以同时得到保障。”小莲思考后自信地做出了回答。

“小莲同学回答得非常好，完全正确，你把你刚才说的说慢一点再重复给大家听。”梁老师开始注意到这个积极思考善于动脑的同学了。

小莲的描述让台下的同学终于都恍然大悟了。

“那老师问大家一个问题，是否数据缓存区的数据批量越大越好呢？”

大家积极思考了一番，觉得如果出问题由于磁盘中的日志文件可恢复，安全可不必担忧了，而在性能方面，也想不出更大批量有啥坏处，余太太还存在一个资金周转的问题，Oracle 应该不存在吧，不过由于感觉梁老师这样的提问肯定有玄机，大家都不说话了。

“同学们，你知道老师为什么上课每次都喜欢用启发性的提问吗？就是希望大家养成思考的习惯，学会多角度看待问题。

在数据库运行过程中，批量刷出的数据占数据缓存区的比例越大，效率一般来说是越高，而且也不用担心断电后的恢复问题。可是大家想想看，如果批量刷出的数据占数据缓存区的比例很大，那断电后数据库重启的恢复数据动作必然需要更长，大家等待的时间也就更长，难道不是这样吗？

因此很多时候要考虑一个平衡问题：批量刷出的量比较小，Oracle 性能就会降低，但是断电开机恢复的时间就更短；反之批量刷出的量比较大，Oracle 性能是更高了，但是断电开机恢复的时间也更长。

此外，我要给大家介绍一个后台进程的新成员，他就是 CKPT。什么时候将数据缓存区数据写到磁盘的动作正是由进程 CKPT 来触发的，CKPT 触发 DBWR 写出。这是我继 DBWR、LGWR 后介绍的第三个重要后台进程，如图 2-16 中圆圈标记处。

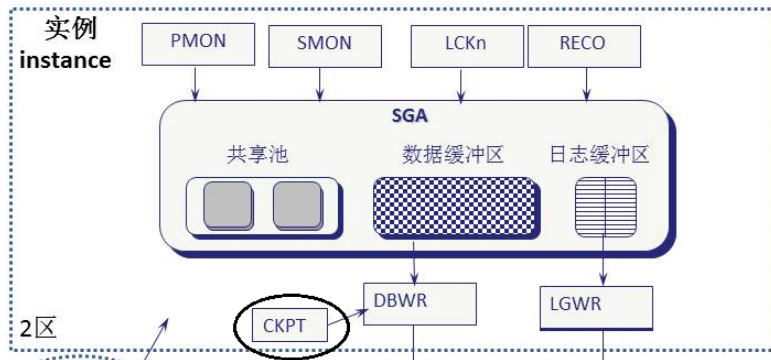


图 2-16 CKPT

这是一个相当重要的进程，我们可以通过设置某参数调整来控制 CKPT 的触发时间，比如万一出现数据库崩溃，希望 Oracle 的 SMON 最多用多长时间来做 Instance recovery，该参数就是 FAST START MTTR TARGET，通过调整该参数，Oracle 会调配 CKPT 在适当的时候去调用 DBWR……当然，这个参数也并非越小越好，太小的数值会导致 Oracle 性能降低。

没想到一个 COMMIT 还是有玄机的，小莲越听越觉得有意思。

“那我再问一个问题，之前描述的 update t set object_id=92 where object_id=29 语句执行完毕后，如果一直不提交，最终会从数据缓存区刷进磁盘吗？”

“会，因为 DBWR 将数据缓存区数据写到磁盘，不是由 COMMIT 决定的，而是由 CKPT 进程决定的。”小莲回答得很自信，台下同学们纷纷表示赞同。

“回答得非常好，我补充一下，在 CKPT 的触发或者说命令下，DBWR 将数据缓存区数据写到磁盘，但是如果 LGWR 出现故障了，DBWR 此时还是会不听 CKPT 的命令罢工的，因为 Oracle 在将数据缓存区数据写到磁盘前，会先进行日志缓冲区写进日志文件的操作，并耐心地等待其先完成，才会去完成这个内存刷到磁盘的动作，这就是所谓的凡事有记录。

好了，大家休息 10 分钟，下一堂课我们一起评选一下数据库后台进程中，谁是劳模。”

劳模？小莲有些发愣。“管他呢，先休息再说。”小莲起身出去休息了。

2.2.3.3 劳模的评选

“欢迎大家来到劳模评选的现场。”休息结束后从梁老师嘴里蹦出的这句话让大家乐了。

“参加评选的总共有 8 名选手，他们分别是 PMON、SMON、LCKn、RECO、CKPT、DBWR、LGWR、ARCH”，他们都有一个共同的特点，都是 Oracle 的后台进程，分别位于体系结构图的如下位置，我们用椭圆框标记出来（如图 2-17 所示）。

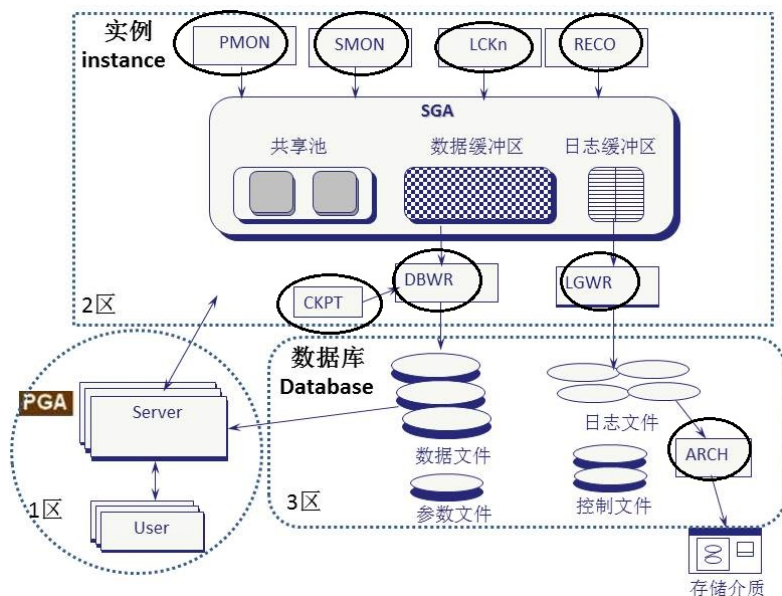


图 2-17 劳模

实际情况是，Oracle 的后台进程远不止这 8 个，我们只需挑选最核心、最重要、最有用的进程来描述就可以了，请大家记住。

要选劳模首先就得了解他们的工作岗位职责及先进事迹，下面我依次介绍每位参评人员的情况。

PMON 的含义为 Processes Monitor，是进程监视器。如果你在执行某些更新语句，未提交时进程崩溃了，这时候 PMON 会自动回滚该操作，无须你人工去执行 ROLLBACK 命令。除此之外还可以干预后台进程，比如 RECO 异常失败了，此时 PMON 会重启 RECO 进程，如果遇到 LGWR 进程失败这样的严重问题，PMON 会做出中止实例这个激烈的动作，用于防止数据错乱。

SMON 的含义为 System Monitor，理解为系统监视器，与 PMON 不同的是，SMON 关注的是系统级的操作而非单个进程，重点工作在于 instance recovery，除此之外还有清理临时表空间、清理回滚段表空间、合并空闲空间，等等。

LCKn 仅使用于 RAC 数据库，最多可有 10 个进程（LCK0，LCK1，…，LCK9），用于实例

间的封锁。

RECO 用于分布式数据库的恢复，全称是 Distributed Database Recovery，适用于两阶段提交的应用场景。这里我简单描述一下，比如我们面临多个数据库 A、B、C，某个应用跨越三个数据库，在发起的过程中需要 A、B、C 库都提交成功，事务才会成功，只要有一个失败，就必须全部回滚。

这和 LCKn 一样，适用的场景比较特殊。

CKPT 进程在前面已经介绍过了，由 Oracle 的 FAST_START_MTTR_TARGET 参数控制，用于触发 DBWR 从数据缓冲区中写出数据到磁盘。CKPT 执行越频繁，DBWR 写出越频繁，DBWR 写出越频繁越不能显示批量特性，性能就越低，但是数据库异常恢复的时候会越迅速。

DBWR 是 Oracle 最核心的进程之一了，负责把数据从数据缓存区写到磁盘里，该进程和 CKPT 相辅相成，因为是 CKPT 促成 DBWR 去写的。不过 DBWR 也和 LGWR 密切相关，因为 DBWR 想将数据缓存区数据写到磁盘的时候，必须通知 LGWR 先完成日志缓冲区写到磁盘的动作后，方可开工。

再说说 LGWR 吧，这个进程的目的很简单，就是把日志缓存区的数据从内存写到磁盘的 REDO 文件里，完成数据库对象创建、更新数据等操作过程的记录。这个 REDO 的记录非同小可，可以用来做数据库的异常恢复，只要保护好了这些 REDO 文件和后续对应的归档文件，从理论上来说，即使数据文件被删除光了，还是可以让数据库根据这些日志记录，把所有的在数据库中曾经发生的事情全部重做一遍，从而保证了数据库的安全。

正因为日志文件对数据库如此重要，LGWR 也成了和 DBWR 一样核心的数据库进程。

因为发生的事情是有顺序的，所以必须顺序地记录。比如数据库的操作是①先建一张表；②插入部分记录；③修改记录；④删除部分记录；⑤增加表字段……

如果日志记录成①，③，②，④，⑤，那恢复的时候就有问题了，会提示数据不存在无须修改。变成②，③，①，④，⑤就更糟糕了，会提示表对象不存在。在保证顺序记录的前提下，出现中断也是绝对不允许的，只会恢复到断号前的场景。比如不小心在记录②的时候失败了，变成只记录了①，③，④，⑤，那数据库最多只能恢复到①，即建表。”说到这里梁老师忽然停下来，问大家为什么。

“因为记录都没插入，还怎么修改删除啊。”同学们因为听得很仔细，所以回答得很迅速。

“那如果是只记录了②，③，④，⑤呢？”梁老师又发问。

“那就更不能恢复了，因为表都没建起来，插数据、改数据、删数据都没意义了。”同学们纷纷抢着回答。

“嗯，所以，有的时候，辛辛苦苦保留了几百个归档日志文件用于恢复，第一个被破坏，保留的那几百个归档文件全都没意义了，都变成废品了。”梁老师补充强调了一句。

小莲听后，不由得心中感叹了一番。

“继续说 LGWR 吧，LGWR 必须记录下所有从数据缓存区写进数据文件的动作，工作任务相当繁重。由于要顺序记录情况下保留的日志才有意义，多进程难以保证顺序，因此 LGWR 只能采用单进程。为了解决这个问题，LGWR 给自己施压，制定了 5 条严格的制度来要求自己，以此来适应工作高强度的日志记录工作。

- ① 每隔三秒钟，LGWR 运行一次。
- ② 任何 COMMIT 触发 LGWR 运行一次。
- ③ DBWR 要把数据从数据缓存写到磁盘，触发 LGWR 运行一次。
- ④ 日志缓冲区满三分之一或记录满 1MB，触发 LGWR 运行一次。
- ⑤ 联机日志文件切换也将触发 LGWR。

最后登场的是 ARCH 进程，它的作用是在 LGWR 写日志写到需要覆盖重写的时候，触发 ARCH 进程去转移日志文件，复制出去形成归档日志文件，以免日志丢失，之前已经描述过了。

好了，现在我的 8 位参评人员都已经介绍完毕，请大家投票选举出一名劳模。”

“LGWR! LGWR! LGWR! LGWR……”台下就听到一种声音。

“好，今天的劳模评选结束了，恭喜 LGWR!”梁老师一本正经的样子逗乐了台下所有的同学。

2.2.3.4 回滚的研究

“在选劳模的过程中，大家了解了 8 位参评人员的先进工作事迹，也顺道明白了 Oracle 主要后台进程的功能和特色，大家都记住了吗？”

“记住了!”小莲回答得最响亮。

“我在给大家描述体系物理结构时开始用的是一条查询语句，结果很快就把体系结构说完了，大家却发现很多体系结构的组件未被提及，所以大家在考虑问题时一定要全面，如果整个数据库只有查询没有任何更新，那 LGWR 就没有存在的意义了，也拿不到劳模的殊荣了。不过这里还是有必要提一下，Oracle 数据库只要启动，就会开始触发各种操作，即使用户不主动读写，系统进程也要有操作，联机日志文件中就会不断记录内容。

说起更新语句，这可是非常关键的 SQL 指令，因为更新语句改变了数据库的数据，正确与否非常重要，所以需要有一个用户确认的过程。如果用户认为更改没问题，就可以发出 COMMIT 指令，放心地将改变的结果保存在数据库中。如果当场后悔了，就发出 ROLLBACK 指令，及时撤销刚才的操作，这就是更新语句的回滚。

比如大家执行 `update t set object_id=92 where object_id=29` 后，如果继续执行一个 ROLLBACK，数据库依然不会将 `object_id=29` 的数据更改为 92，这中间到底发生了什么，Oracle 是如何做到的呢？”梁老师说到这里有些故作神秘，稍稍停顿了一会儿。

台下静悄悄的。

“那我来说说这个语句的回滚过程吧，为了简化描述，前面说过的 PGA 和共享池区的经历我

就不再重复了。

- ① 想更新 `object_id=29` 的记录首先就需要查到 `object_id=29` 的记录，检查 `object_id=29` 是否在数据缓存区里，不存在则从磁盘中读取到数据缓存区中，这一点和普通的查询语句类似。
- ② 但是这毕竟不是查询语句而是更新语句，于是要做一件和查询语句很不同的事，在回滚表空间的相应回滚段事务表上分配事务槽，从而在回滚表空间分配到空间。该动作需要记录日志写进日志缓存区。
- ③ 在数据缓存区中创建 `object_id=29` 的前镜像，前镜像数据也会写进磁盘的数据文件里（回滚表空间的数据文件），从缓存区写进磁盘的规律前面已经说过了，由 CKPT 决定，当然也别忘记了这些动作都会记录日志，并将其写进日志缓存区，劳模 LGWR 还在忙着将日志缓存区的数据写入磁盘形成 redo 文件呢。
- ④ 前面步骤做好了，才允许将 `object_id=29` 修改为 `object_id=92`，这个显然也是要记录进日志缓存区的。
- ⑤ 此时用户如果执行了提交，日志缓存区立即要记录这个提交信息，然后就把回滚段事务标记为非激活 INACTIVE 状态，表示允许重写。
- ⑥ 如果是执行了回滚呢，Oracle 需要从回滚段中将前镜像 `object_id=29` 的数据读出来，修改数据缓存区，完成回滚。这个过程依然要产生日志，要写数据进日志缓存区。”

说到这里，梁老师忽然停了下来，丰富的授课经验告诉他，学员学习每一段原理如果没有积极思考，是很难真正理解或者说很容易就淡忘了的。

“同学们，老师先说到这里，大家都听明白了吗，有什么疑问就举手问吧。”

“梁老师，记录前镜像为什么也要写日志，回滚的东西干嘛要记录，我们偶尔才需要回滚吧。”胖小伙子举手发问。

“那你认为前镜像记录首先是记录在哪里，SGA 的数据缓存区中，还是磁盘的回滚段的数据文件里？”梁老师笑着问。

“数据缓存区里。”

“正确，那 SGA 的数据缓存区中的前镜像数据什么时候会刷新到磁盘里呢？”

“当数据缓存区的数据达到一定量的时候，由 CKPT 触发数据缓存区写出刷新到磁盘中去，前镜像数据和正常的数据库操作应该是一样的吧。”小莲看到胖小伙子暂时回答不出来，忍不住抢答了。

“小莲回答得非常好，其实刚才那位同学的疑问主要就是在没有意识到用于准备回滚的前镜像数据的生成其实和普通数据库操作差不多，唯一的差别就在于一个是刷新到磁盘的普通文件里，一个是刷新到磁盘的回滚数据文件里。

如果这样，就好理解了。普通数据库操作可能会出现事务已经 COMMIT 了，但是数据在数据

缓存区并没有立即刷新到磁盘，数据丢失后需要依据 redo 来重做的场景。回滚前镜像数据也是如此，你也需要日志记录相关前镜像的操作来应对万一用户需要回滚的情况，否则当回滚的前镜像数据既不在内存又不在磁盘的情况出现后（比如突然断电等），用户此时该如何做回滚呢？当然是依据记录了前镜像相关操作的日志来重新做一次还原前镜像的操作。”

同学们这下听明白了。

“接下来大家跟我一起看看回滚段的相关参数，其中 UNDO_MANAGEMENT 为 AUTO 表示是自动回滚段管理，回滚段空间不够时可以自动扩展；UNDO_RETENTION 为 900 的含义是，DML 语句需要记录前镜像，当 COMMIT 后，表示回滚段保留的前镜像被打上了可以覆盖重新使用的标记，但是要在 900 秒后方可允许；UNDO_TABLESPACE 为 UNDOTBS1 就不用多解释了，表示回滚段表空间的名字为 UNDOTBS1，具体如下：

```
SQL> show parameter undo
```

NAME	TYPE	VALUE
undo_management	string	AUTO
undo_retention	integer	900
undo_tablespace	string	UNDOTBS1

为了加深大家的理解，我想考大家一个问题，DML 语句一般分为三类，即 insert 插入、update 修改和 delete 删除，大家觉得这三类语句哪种语句对回滚的相关操作负荷最大，哪种最小。或者说哪种操作产生的 undo 最多，哪种产生的 undo 最少？”

同学们沉默了许久，梁老师开始提醒大家：“回滚记录了什么？是记录了反向操作的动作，用于后续执行后覆水可收的。”梁老师修改的这个成语把大家再次逗乐了。

“我明白了，应该是 delete 最多吧，删除的反向操作是插入，等于是整行记录完整地插入回去，那就是要把表的所有字段信息都记录下来。insert 应该最少吧，反向操作是 delete，这个只要标记定位到原先的位置就可以，是不是只要记录这个定位的标记就可以了？而 update 居中，只需要记录更新前字段的值并保存起来就可以了。”晶晶回答得相当流利。

“回答得非常好，更准确的说法是 insert 的 undo 信息是记录了插入记录的 rowid，这个就是你说的唯一标记，根据这个 rowid 就足以定位到插入的记录并删除，从而达到回退目的。rowid 在未来的学习中我们会详细描述。”

台下同学们不住地点头。

“那我再问第 2 个问题，我们不说 undo 说 redo，insert、update、delete 三类语句，哪种记录 redo 最多，哪种最少？”

“梁老师，这个和 undo 正好相反啊，redo 就是把刚才做的事情重新再做一遍，那 delete 只需要记录相关 rowid 就可以再删除一遍了，很精炼，产生的 redo 最少，而 insert 如果想再完成一遍，至少需要知道所有字段的取值，产生的 redo 肯定最多啊。不过 update 的情况不变，都是居中。”

小莲回答得很自信，她知道自己一定回答正确了，经过梁老师启发性的发问后，她感觉自己对于回滚的理解加深了很多。

“这个回答显然是经过思考的，很不错，不过我先不说对不对，再问第 3 个问题，专门针对 undo 会产生对应的 redo 吗？”

“会！”沉默了一小会儿，台下有几个同学同时大声回答。

“很好，梁老师前面说过了，undo 的信息是准备用户回退的前镜像信息，是用于大家后悔自己操作后的还原动作，这些数据也是需要保护的，如果丢失了我们就别想还原了，而 redo 是非常好的恢复宝物，有他的记录我们就放心了。

下面我总结一下 DML 语句的一个特点：

DML 语句不同于查询语句，会改变数据库的数据。除此之外，还会产生用于将来恢复的 redo 和用于回退的 undo。另外还有一个细节就是，由于 undo 也需要保护，所以还会专门产生保护 undo 操作的 redo。”

小莲觉得自己不仅是听明白了，思路也开阔了，她觉得 Oracle 的机制还挺合理的。

“大家还记得我刚才没有答复小莲的回答是否正确，大家好好思考一下，我不做回答，把这个悬念留在后头，在后面表设计的章节中会有相关试验，那时大家就会明白答案了。”

2.2.3.5 一致的查询

“同学们，前面有关回滚的机制都听明白了吧，无论是 undo 还是 redo，都和更新语句有关，和查询语句无关。

现在大家再回顾一下我之前最早举例的查询语句 `select object_name from t where object_id=29`，我对这个语句的执行情况做一个极端的假设，就是假设该语句查询时间非常长，我从早上 8 点开始发起查询，直到早上 9 点才查询完毕，返回结果给我。

可是 8 点到 9 点这段时间里，数据库发生了很多变化，比如 `object_id=29` 的记录 8 点的时候是可以返回 2 条结果的，但是 8 点半的时候已经被别的用户删除甚至还提交了，请问 9 点返回结果时，我看到的记录是没有返回还是返回 2 条，换句话说，是查询到 8 点那个时间点数据库的情况，还是 9 点这个时间点数据库的情况？”

同学们迟疑地你看我，我看你，有些犹豫不决，不过大部分回答都是查询不到记录，即返回的结果是基于 9 点那个时间点数据库的真实情况的结果。

“看来大家基本都认为该 SQL 语句会返回 9 点结束查询时数据库的数据。到底是对还是不对呢？”

“梁老师，我觉得这个没有对还是不对啊，都有道理啊，要求返回 8 点那个时刻 T 表结果的人，认定自己要的是查询时刻的 T 表的记录情况。而认定返回 9 点那个时刻 T 表的人，他查询的时候是返回了当前数据库的真实情况，感觉各自都有各自的道理啊。”刚才一言不发的小莲起身说道。

“小莲说得很好，我发现大多数人还是更倾向于查询结束后，T 表此时情况要真实反映给自己，好比有人查询自己有没有被录用，从 8 点查到 9 点才结束，但是自己的录用记录是在 8 点半时被插入的，如果 9 点查询结束后返回结果发现自己被录用了，很开心，但是如果是按 8 点时刻来查询，最终只是失望地离去，实际上他已经被录用了。

现在大家觉得返回 8 点的和返回 9 点的表记录这两种机制，哪种更合理，选哪个？”

“9 点！”这下大家回答得异口同声。

“哦，刚才我随便给大家举了一个在数据库中查询自己录用情况的例子后，大家都变得很肯定了，那我再给大家说一个故事，看看大家会不会改变想法。

比如我有 4 个户头，户头 1 金额为 1000 元，户头 2 为 2000 元，户头 3 为 3000 元，户头 4 为 4000 元，如果我在数据库中查询自己的 4 个账户的总金额是多少，应该是返回 10000 元，细节为：1000+2000+3000+4000=10000 元。具体如图 2-18 所示。

户头1
1000元
户头2
2000元
户头3
3000元
户头4
4000元

图 2-18 4 个户头

假如数据库查询返过程很慢，当我从自己的户头 1 读到户头 3，还没来得及读到户头 4 时，我家人从户头 1 转移 500 元到户头 4 上，从而户头 1 变成 500 元，户头 4 变成 4500 元，其实此时我的总金额依然不变还是 10000 元，细节为：500+2000+3000+4500=10000 元，如图 2-19 所示。

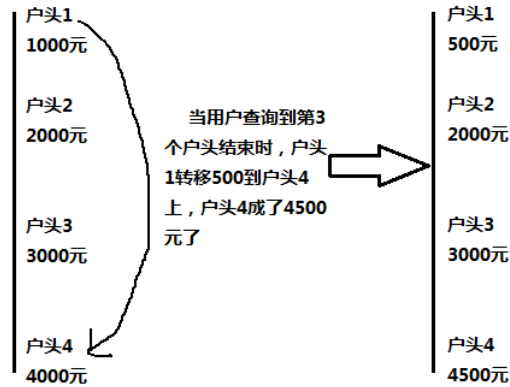


图 2-19 转户头

但是实际情况是，由于户头 1 到户头 3 我都已经读过了，Oracle 不可能实时回头读以前读过的数据，否则查询永远都结束不了了。

因此情况就有可能变成这样，户头 1 到户头 3 读过时是 $1000+2000+3000$ ，而读到户头 4 时该户头变为 4500，总金额就是 $1000+2000+3000+4500=10500$ ，这显然是一个错误的答案，我的户头金额总数怎么会变成 10500 呢？（如图 2-20 所示）

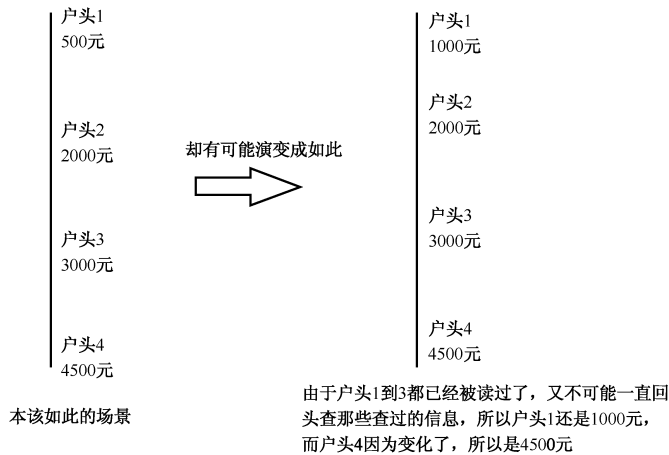


图 2-20 转户头后有错

梁老师查询自己所有户头总金额时发现结果展现是 10500 元，比预计的多了 500，这多出来的 500 元是不是要用来请大家吃饭啊？”

台下引来笑声一片，同学们都乐了。

“同学们知道问题出在哪里吗，如果我查询到户头 4 的时候，不是 4500 而是 4000，结果是对

的还是错的？”

“是对的！” 同学们都看得很清楚。

“那刚才我说的 8 点查询开始到 9 点查询结束，返回的结果是 8 点的结果还是 9 点的，现在可以回答了吗？” 梁老师又重复了一下引导大家思考之前提出的问题。

“8 点！” 所有同学都改口了。

“很好，如果老师查询自己户头总金额的那一刻起，中间的变化都不去理会，那老师就不会多出 500 元来请大家了，这就是所谓的一致读。

查询的结果由查询的那个时刻决定了，数据新的变化是不予理睬的。如果不这样，Oracle 居然会得出我的账户是 10500 这个无论何时都不可能存在的错误数据，所以 Oracle 务必要保持一致读，避免错误产生。现在大家都明白了吗？”

“这梁老师说得太真是通俗易懂啊，还真没细想数据库会存在这样的情景，看来设计数据库软件给大家使用，需要考虑的东西还真不少啊。” 小莲心中暗自想到。

“大家再想想另外一个问题，Oracle 是如何保证查询到查询的那个时间点的数据的，比如大家知道，我查到最后一个户头的时候，明明这个值是 4500，我却一定要查出 4000，否则就会有问题，那数据库是如何做到的呢？”

台下半晌没人答上话来，是啊，Oracle 如何做到这点呢？小莲也没想明白。

“其实这和回滚段有关系，Oracle 的回滚段不仅仅是保证了数据的回退，其实还提供了另外一个功能，保证数据库的一致读。

回滚段究竟是如何保证了数据库的一致读的呢？这其中到底隐藏着什么不为人知的秘密呢……大家休息 10 分钟，不要走远，后面的节目更精彩！”

2.2.3.6 一致读的原理

“大家好，现在我们来谈一致读的原理，就是查询的记录由查询的这一时间点决定，后面即便变化了，也要根据回滚段保存的前镜像记录，取到那个时间点的数据。刚才梁老师举了查询自己账户的例子，大家应该印象很深刻吧。

不过还有一个细节，比如我的转账记录是发生在我查询之前，由另一个人操作的，但是没提交，此时我依然看不到这个转账的变化。这个该如何保证呢？下面我也会一并说说。

我想把问题说明白，必须先知道下面两个前提。

- ① 了解数据库的 SCN，SCN 的全称是：System Change Number，这是一个只会增加不会减少的递增数字，存在于 Oracle 的最小单位块里，当某块改变时 SCN 就会递增。
- ② 数据库的回滚段记录事务槽（前面我在描述回滚的时候提过，事务槽是用来分配回滚空间的），如果你更新了某块，事务就被写进事务槽里。如果未提交或者回滚，该块就存在活动事务，数据库读到此块可以识别到这种情况的存在。

现在我们来描述一下 Oracle 是如何实现一致读的，还是举我刚才转账的例子，当我早上 8 点整开始查询数据库时，首先会获取 8 点那个时刻的 SCN 号，并记录下来，比如是 SCN8:00，那么

8 点的 SCN8:00 一定大于等于记录在所有数据块头部的 ITL 槽中的 SCN 号（如果有多个 ITL 槽，则为其中最大的那个 SCN 号）。比如在数据库中查询我的 4 个账户经历了块 1 到块 4，而 8 点后需要查询的这些块都没更新，结果在查询过程中，发现这些块的 SCN 其实都小于 SCN8:00，说明该块在这段时间内确实没被更新过，我们就放心地全读进去，总金额就是 $1000+2000+3000+4000=10000$ ，如图 2-21 所示。

假如 8 点以后有更新，发生转账动作，那是怎么回事呢？比如当 8 点 30 分查询到块 3（户头 3）刚结束时，户头 1 转 500 元到户头 4，户头 1 从 1000 元变成 500 元，户头 4 从 4000 元变为 4500 元。此时块 1 和块 4 的 ITL 槽中的 SCN 号同时改变了，都变为 SCN8:30，当数据库查询到块 4 时，发现块 4 的头部的 ITL 槽中的 SCN 号 SCN8:30 大于发出查询时间的 SCN8:00，说明该数据块在 8 点以后被更新了，于是根据 ITL 槽中记录了对应的 undo 块的地址找到 undo 块，将 undo 块中的被修改前的数据（4000）取出，从而构建出被更新之前的那个时间点（8 点 30）的数据块内容，这个值就是 4000。

以下是查询开始到结束数据库块未改变的场景
8点开始查询，系统SCN为SCN8:00

户头1 1000元	块1 SCN7:00
户头2 2000元	块2 SCN6:00
户头3 3000元	块3 SCN7:00
户头4 4000元	块4 SCN7:30

图 2-21 转户头与 SCN

此外虽然块 1 也变化为 SCN8:30 了，不过由于之前已经读过了，所以不会回头去比较，所以记录的还是 1000，具体见图 2-22。

收获，不止 Oracle

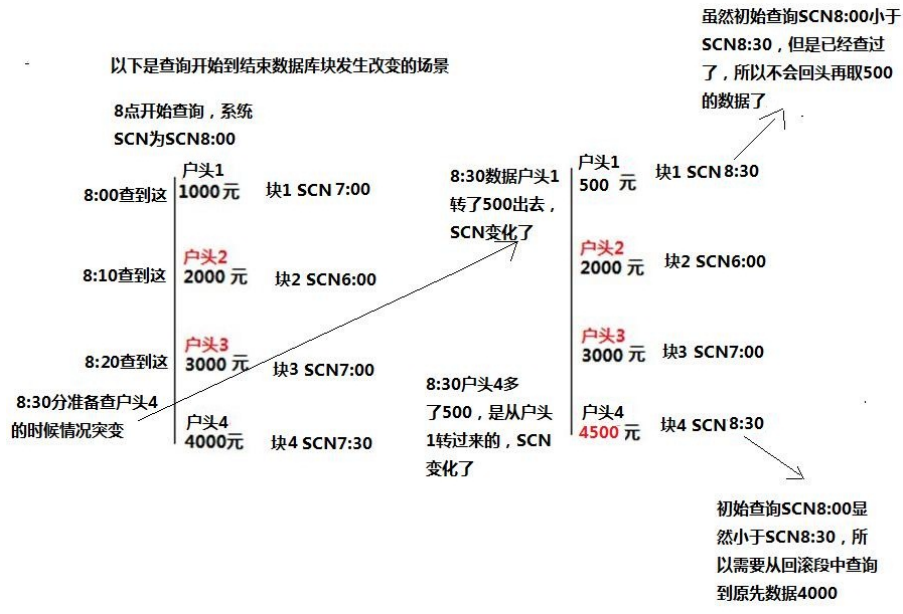


图 2-22 SCN 详细分析

因此我们查询的结果还是 $1000+2000+3000+4000=10000$ 。

如果我们再读一遍数据库，比如 9 点 10 分开始，此时系统取到的 SCN 为 SCN9:10，这时块 1 的 SCN8:30，块 2 的 SCN6:00，块 3 的 SCN7:00，块 4 的 SCN8:30，每个都小于 SCN9:10，所以此时的总和是 $500+2000+3000+4500=10000$ ，和上一次的 $1000+2000+3000+4000$ 不一样了，但是结果却都是对的，都是 10000。

不过并不是查询开始的 SCN 大于等于查询中所有块的 SCN 就一定可以直接获取数据，什么情景是这样的呢？”

梁老师辛苦描述了大半天，终于停下来问台下的同学们了。

“已经读过的块 SCN 如果大于当前查询的 SCN，因为查过的不会回头查。”小胖子起身回答。

“很好，你说得很对，但是梁老师想说的不是这个，是指还没读过的场景，刚读到，发现 SCN 小于我们发起查询的 SCN，不过还是不读取这个块的数据，去回滚段读取，是说这个。”

“会不会是别的用户发起的，更新未提交的数据啊？”小莲举手回答。

“正确，比如梁老师是 8 点开始查询的，7 点时我的户头 1 为 1000，7 点 30 分被人更新为 500 未提交，此时我开始查询，SCN8:00 显然大于 SCN7:30，不过 Oracle 会发现另外一个问题，就是这个块有活动事务，所以还会从回滚段找到之前的前镜像的数据，还是 1000，此外要特别注意，如果由于前镜像被人不断地重写，1000 这个数据从回滚段里找不回来了，那 Oracle 这个查询将会以 ORA-01555 的报错而中止退出，你可以查询失败，但是不可以查询出一个错误的结果来。

因此 Oracle 在做一致读时，首先是看发起的 SCN 是否大于当前查询块的 SCN，如果小于，毫无疑问从回滚段获取前镜像数据。如果 SCN 确实大于当前查询块的 SCN，还要确保该块没有活动事务，否则还是要去前镜像查找。如果更新梁老师账户的那个人在 7:30 操作完后提交或者回退，那该块就不存在活动事务了。

现在大家听明白了吗？”

“明白了！”同学们回答得很一致，这回大家认识得比较清晰了。

“早期的 SQL Server 的数据库版本，是读产生锁，在读数据时表就被锁住，这样确实是不存在问题了，不过如果读表会让表锁住，那数据库的并发也就做得太糟糕了。早期的其他数据库版本也有边读边锁的，比如已经读过的记录就允许被修改，而未读过的数据却是被锁住的，不允许修改，这虽然稍稍有些改进，只锁了表的部分而非全部，但是还是读产生锁，非常糟糕。

而 Oracle 的回滚段，却解决了读一致性的问题，又避免了锁，大大增强了数据库并发操作的能力。”

这下大家对 Oracle 的回滚段有了新的认识，原来既可以回滚数据，又可保证一致读，小莲觉得受益匪浅，同时也感叹软件行业设计的重要性。

2.2.3.7 实践的体会

“同学们，至此我已经将 Oracle 的体系结构基本说完，大家都听懂了吗？”

“梁老师，听是听懂了，不过觉得有一点点抽象，不够直观啊。”小莲起身回答。

“还不够直观啊？梁老师又是画图又是讲故事，还执行 SQL 指令让大家去感受 Oracle 设计的优良之处，大家还记得 SQL 语句先后两次执行效率差别很大吧，知道原因吗？”

“知道！”大家响应倒是非常积极。

“其实我知道你们的意思。”梁老师笑着说，“你们只是感觉我说的这些东西看不见摸不着，都只是我说说而已，想眼见为实，是吧？”

台下同学纷纷点头。

1. 内存的体会

“大家脑子里再回想一下体系结构图，梁老师描述了 SGA、PGA 内存区，首先体会这个。

梁老师这里有一个数据库环境给大家体会一下，sqlplus "/ as sysdba" 连进数据库后，我们可以用 show parameter sga 来了解 SGA 开辟多大空间，用 show parameter pga 来知道 PGA 开辟多大空间，具体如下：

```
[oracle@itmapp3 ~]$ sqlplus "/ as sysdba"
```

```
SQL> show parameter sga
```

NAME	TYPE	VALUE
lock_sga	boolean	FALSE

```
pre_page_sga                boolean    FALSE
sga_max_size                big integer 2368M
sga_target                  big integer 2368M
SQL> show parameter pga
NAME                        TYPE      VALUE
-----
pga_aggregate_target        big integer 788M
```

脚本 2-4 查看 SGA 及 PGA 的分配大小

这里说明 SGA 是 2368M，而 PGA 是 788M。那共享池和数据缓冲区又是多大呢？
我们继续往下查看：

```
SQL> show parameter shared_pool_size
NAME                        TYPE      VALUE
-----
shared_pool_size            big integer 0
SQL> show parameter db_cache_size
NAME                        TYPE      VALUE
-----
db_cache_size                big integer 0
```

脚本 2-5 查看共享池和数据缓冲池的分配大小

结果令人大吃一惊，共享池和数据缓存区的大小都为 0，这是咋回事呢？这是因为这里 Oracle 设置为 SGA 自动管理，共享池和数据缓存区的大小分配由之前的 SGA_MAX_SIZE 和 SGA_TARGET 决定，总的大小为 2368M，它们分别被分配多少由 Oracle 来决定，无须我们人工干预，其中 SGA_TARGET 不能大于 SGA_MAX_SIZE。二者有什么差别呢？举个例子，比如 SGA_TARGET=2G，而 SGA_MAX_SIZE=8G，表示数据库正常运行情况下操作系统只分配 2G 的内存区给 Oracle 使用，而这 2G 就是共享池和数据缓存区等内存组件分配的大小，可是运行中发现内存不够用，这时 OS 可以再分配内存给 SGA，但是最大不可以超过 8G。

一般情况下都建议使用 SGA 内存大小自动分配的原则，如果一定要手工分配也行，把 SGA_TARGET 设置为 0，再把 SHARED_POOL_SIZE 和 DB_CACHE_SIZE 设置为非 0，就可以了。

此外我们这里是以 Oracle 10g 为例，在 Oracle 11g 中，自动化程度更彻底，推出了 MEMORY_TARGET 参数，只要设置这个参数值，连 PGA 都不需要设置了，MEMORY_TARGET 参数指定的内存会自动分配内存给 SGA 与 PGA。

接下来我要让大家看得更直观、更具体一点，大家看如下 **ipcs -m** 的查看共享内存的命令：

```
[oracle@itmapp3 ~]$ ipcs -m
----- Shared Memory Segments -----
```

key	shmid	owner	perms	bytes	nattch	status
0x00020001	0	nmtestv3	666	12	8	
0x66a02988	2097153	oracle	660	2485125120	23	
0x110280be	1736708	agentv3	666	136	1	
0x110280ba	1769477	agentv3	666	30720	1	
0x00000000	1081351	oracle	600	393216	2	dest
0x00000000	1114120	oracle	600	393216	2	dest
0x00000000	1146889	oracle	600	393216	2	dest
0x00000000	1179658	oracle	600	393216	2	dest
0x00000000	1212427	oracle	600	196608	2	dest
0x00000000	1245196	oracle	600	393216	2	dest
0x00000000	1277965	oracle	600	393216	2	dest
0x00000000	1310734	oracle	600	196608	2	dest
0x00000000	1343503	oracle	600	393216	2	dest

脚本 2-6 从操作系统层面来感受 SGA 分配情况

我们看看 Oracle 开辟的共享内存到底是不是 2368MB，其中 dest 表示共享内存段已经被删除，但是仍然有程序在连接着它，所以 Oracle 开辟的共享内存段大小是 2485125120，换算成 MB 是 $2485125120 / 1024 / 1024 = 2370\text{MB}$ ，基本上等同于 2368MB，略有一点点误差是正常的。

好了，梁老师把 Oracle 的内存区说完了，大家有什么疑问吗？”

“梁老师，我觉得不对啊，日志缓存区您好像没说啊，这个大小也是自动分配吗？”晶晶举手问。

“非常好！老师发现大家上课都非常认真，而且记忆力很强，我特意漏说了这个，居然这么快就被同学们揪出来了。让我们看看日志缓存区的大小在我这个数据库环境中分配了多少。

```
SQL> show parameter log_buffer
```

NAME	TYPE	VALUE
log_buffer	integer	14242816

脚本 2-7 查看日志缓冲区的分配大小

这里可以看出，log_buffer 分配的大小大致为 15MB，大家知道，log_buffer 的大小是不能由 SGA_TARGET 来自动分配的，这个必须手动分配和调整。由于 log_buffer 每满 1MB 就要写一次，每满三分之一也要写一次，所以分配太大优化效果也不是很明显，一般 15MB 即可满足需求。

大家听到这里，还有什么疑问吗？”

“梁老师，这些参数的设置需要我们如何修改啊？”同样很积极的胖小伙子敬昱起身提问。

“敬昱同学问得很好，如果老师在上课，学生只是听课记录而没有一点自己的想法，那效果根本就不会明显。这里显而易见的是，如果我们不修改数据库的这些内存参数，那所有的取值都

是默认的，默认的取值可以满足所有的项目需求吗？肯定是不可能的，所以学会修改数据库参数就变得很重要了。具体命令如下。

ALTER SYSTEM 命令增加了一个新的选项 scope。scope 参数有 3 个可选值：memory、spfile 和 both。

- memory: 只改变当前实例运行，重新启动数据库后失效。
- spfile: 只改变 spfile 的设置，不改变当前实例运行，重新启动数据库后生效。
- both: 同时改变实例及 spfile，当前更改立即生效，重新启动数据库后仍然有效。

可以通过 ALTER SYSTEM 或者导入导出更改 spfile 的内容。

针对 RAC 环境，ALTER SYSTEM 还可以指定 SID 参数，对不同实例进行不同的设置。

通过 spfile 修改参数的完整命令如下：

```
alter system set <parameter_name>=<value> scope=memory|spfile|both [sid=<sid_name>]
```

- ① 如果当前实例使用的是 pfile 而非 spfile，则 scope=spfile 或 scope=both 会产生错误；
- ② 如果实例以 pfile 启动，则 scope 的默认值为 memory，若以 spfile 启动，则默认值为 both；
- ③ 有些参数必须重启才能生效，如 log_buffer；
- ④ scope 不写默认是表示 both。

现在我想把 sga_target 从 2368MB 改为 2000MB，这个命令如下：

```
SQL> show parameter sga
```

NAME	TYPE	VALUE
lock_sga	boolean	FALSE
pre_page_sga	boolean	FALSE
sga_max_size	big integer	2368M
sga_target	big integer	2368M

```
SQL> alter system set sga_target=2000M scope=spfile;
```

```
System altered.
```

脚本 2-8 修改 SGA 大小（scope=spfile 方式）

请问同学们，现在修改过来了吗？”

“改过来了！”大家异口同声。

梁老师笑了笑，再输入一次 show parameter sga，结果大家发现 sga_target 依然是 2368M 而不是 2000M。

“应该是要加 scope=memory 或者是 scope=both 才会立即生效！”终于有同学反应过来了。

“很好，memory 重启后修改的参数就失效了，而 both 却可以永久保存。下面我举 scope=both 的例子，如下：

```
SQL> show parameter sga
```

NAME	TYPE	VALUE
lock_sga	boolean	FALSE
pre_page_sga	boolean	FALSE
sga_max_size	big integer	2368M
sga_target	big integer	2368M

```
SQL> alter system set sga_target=2000M scope=both;
System altered.
SQL> show parameter sga
```

NAME	TYPE	VALUE
lock_sga	boolean	FALSE
pre_page_sga	boolean	FALSE
sga_max_size	big integer	2368M
sga_target	big integer	2000M

脚本 2-9 修改 SGA 大小（scope=both 方式）

终于改过来了，现在大家会修改了吧？这里要注意的是 scope=xxx 可以不写，默认为 scope=both，此外 log_buffer 等参数必须重启才会生效，因此 alter system set log_buffer=15000000 scope=memory 或者是 scope=both 就会报错，只支持 alter system set log_buffer=15000000 scope=spfile，然后重启后生效，具体如下：

```
SQL> show parameter log_buffer;
```

NAME	TYPE	VALUE
log_buffer	integer	14242816

```
SQL> alter system set log_buffer=15000000 scope=memory ;
alter system set log_buffer=15000000 scope=memory
*
```

ERROR at line 1:
ORA-02095: specified initialization parameter cannot be modified

```
SQL> alter system set log_buffer=15000000 scope=both;
alter system set log_buffer=15000000 scope=both
*
```

ERROR at line 1:
ORA-02095: specified initialization parameter cannot be modified

```
SQL> alter system set log_buffer=15000000 scope=spfile;
System altered.
```

收获，不止 Oracle

不过查看依然没有更改

```
SQL> show parameter log_buffer;
```

NAME	TYPE	VALUE
log_buffer	integer	14242816

重启数据库后才能变化为 15000000

脚本 2-10 修改 LOG_BUFFER 参数

2. 进程的体会

“大家好，还记得 Oracle 数据库的组成吗？Oracle 数据库是由实例和一组数据库文件组成的，实例则是由 Oracle 开辟的内存区和一组后台进程组成的。

关于后台进程我们之前聊了最主要的 8 个，其中 LGWR 同志还被大家全票通过，当选为劳模，大家都投出自己神圣的一票，应该记忆深刻吧？

不过在描述体系结构的过程中虽然我极力说得生动形象，但难免还是有些抽象。幸亏我及时带大家登录数据库环境将数据库内存体会了一把，现在大家终于对 SGA 和 PGA 有了更形象的认识了。

体验过 Oracle 内存区后，现在我们继续登录 Oracle 数据库环境，来体会一下这些后台进程，我们登录的环境是 Linux/UNIX 环境，因为 Windows 环境中 Oracle 是多线程形式的，不好查看。而 UNIX 环境是多进程形式的，更方便大家查看，如图 2-23 所示。

```
[oracle@itmapp3 ~]$ ps -ef |grep oracle
oracle 14457 1 0 Jul17 ? 00:01:34 ora_pmon_itmtest
oracle 14459 1 0 Jul17 ? 00:00:02 ora_psp0_itmtest
oracle 14461 1 0 Jul17 ? 00:02:08 ora_mman_itmtest
oracle 14463 1 0 Jul17 ? 00:02:05 ora_dbw0_itmtest
oracle 14465 1 0 Jul17 ? 00:15:24 ora_lgwr_itmtest
oracle 14467 1 0 Jul17 ? 00:06:25 ora_ckpt_itmtest
oracle 14469 1 0 Jul17 ? 00:02:22 ora_smon_itmtest
oracle 14471 1 0 Jul17 ? 00:00:00 ora_reco_itmtest
oracle 14473 1 0 Jul17 ? 00:03:47 ora_cjq0_itmtest
oracle 14475 1 0 Jul17 ? 00:05:03 ora_mmon_itmtest
oracle 14477 1 0 Jul17 ? 00:20:42 ora_mmln_itmtest
oracle 14479 1 0 Jul17 ? 00:00:00 ora_d000_itmtest
oracle 14481 1 0 Jul17 ? 00:00:00 ora_s000_itmtest
oracle 14554 1 0 Jul17 ? 00:00:30 ora_qmnc_itmtest
oracle 14598 1 0 Jul17 ? 00:00:03 ora_q001_itmtest
oracle 2396 1 0 09:36 ? 00:00:00 oracleitmttest (LOCAL=NO)
oracle 2584 1 0 13:55 ? 00:00:00 oracleitmttest (LOCAL=NO)
oracle 3607 1 0 13:59 ? 00:00:00 oracleitmttest (LOCAL=NO)
oracle 3972 1 0 14:01 ? 00:00:00 oracleitmttest (LOCAL=NO)
oracle 4036 1 0 14:01 ? 00:00:00 oracleitmttest (LOCAL=NO)
oracle 5684 1 0 Aug15 ? 00:00:00 oracleitmttest (LOCAL=NO)
oracle 7582 1 0 14:17 ? 00:00:00 oracleitmttest (LOCAL=NO)
--略去
```

图 2-23 Oracle 进程体会

方框标记的部分果然是刚才熟悉的几个进程，Oracle 还有很多其他相关进程，这里就不一一探讨分析了，另外这里可以看出 Oracle 实例的名叫 **itmtest**，具体可以在数据库中查看到，如下：

```
SQL> show parameter instance_name
```

NAME	TYPE	VALUE
instance_name	string	itmtest

脚本 2-11 查看 Oracle 实例名

因此我们刚才其实可以如下搜索更准确，即 `ps -ef |grep itmtest`。

此外大家注意 LOCAL=NO 的部分（如图 2-24 所示），这表示是非 Oracle 本身的后台进程，是别的用户通过监听，连进该数据库进行访问的，关于监听，我们在随后会让大家体会到。

oracle	2396	1	0	09:36 ?	00:00:00	oracleitmtest	{LOCAL=NO}
oracle	2584	1	0	13:55 ?	00:00:00	oracleitmtest	{LOCAL=NO}
oracle	3607	1	0	13:59 ?	00:00:00	oracleitmtest	{LOCAL=NO}
oracle	3972	1	0	14:01 ?	00:00:00	oracleitmtest	{LOCAL=NO}
oracle	4036	1	0	14:01 ?	00:00:00	oracleitmtest	{LOCAL=NO}
oracle	5684	1	0	Aug15 ?	00:00:00	oracleitmtest	{LOCAL=NO}
oracle	7582	1	0	14:17 ?	00:00:00	oracleitmtest	{LOCAL=NO}

图 2-24 LOCAL 为 NO 的进程体会

LOCAL=NO 的这些进程如果被杀了，数据库不会崩溃，只是某些应用正好连上来操作数据库，被强制给踢出数据库了，而那些 LGWR 和 DBWR 进程如果被杀了，那数据库立即就崩溃了，操作请千万小心。

不过这里少了一个很重要的进程，就是 ARCH，这是为什么呢？”梁老师忽然停下来问大家。

```
[oracle@itmapp3 ~]$ ps -ef |grep arc
```

```
oracle 16583 16546 0 19:03 pts/0 00:00:00 grep arc
```

脚本 2-12 查看 Oracle 归档进程

台下同学都没想明白，大家纷纷摇头。

“大家都还记得这是归档进程，当日志循环写入过程中会出现下一个日志已经被写过的情况，再继续写将会覆盖其内容，需要将这些即将被覆盖的内容写出到磁盘里去形成归档文件，这样日志记录不会丢失，将来数据库就可以从这些日志文件和归档文件中进行数据库的恢复处理。

不过这个归档并非总是必要的，因为有的数据库只是用来测试的，安全性要求不高，此时就可以考虑把归档给关闭了，数据库少做一件事，效率自然就更高了。

所以刚才同学们查不到归档进程，正是因为被关闭了。”

同学们这下才都明白过来，不过梁老师的启发式教学让大家学会了上课期间活跃思考问题，马上就有同学站起来问梁老师：“怎么看归档是开还是关，我们又如何打开和关闭呢？”

“好，那我就演示给大家看看，首先是如何查看数据库归档是开启还是关闭，其实很简单，就是登录数据库后输入 `archive log list` 命令，具体如下。显而易见，Database log mode 为 No

收获，不止 Oracle

Archive Mode 表示当前数据库是非归档的。

```
[oracle@itmapp3 ~]$ sqlplus "/ as sysdba"
SQL> archive log list;
Database log mode                No Archive Mode
Automatic archival                Disabled
Archive destination              USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence       3670
Current log sequence             3672
```

脚本 2-13 查看 Oracle 归档是否开启

更改数据库归档模式比较麻烦，需要重启数据库，将数据库置于 mount 状态后，输入 alter database archivelog（如果是归档改为非归档，这里是 alter database noarchivelog），然后再开启数据库 alter database open，才可以将数据库更改为非归档，具体步骤如下：

```
SQL> shutdown immediate
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> startup mount;
ORACLE instance started.
Total System Global Area        2097152000 bytes
Fixed Size                      2085192 bytes
Variable Size                   520097464 bytes
Database Buffers                1543503872 bytes
Redo Buffers                    31465472 bytes
Database mounted.
SQL> alter database archivelog;
Database altered.
SQL> alter database open;
Database altered.
```

脚本 2-14 将 Oracle 归档开启方法

现在再看我们发现，数据库已经是归档模式了，具体如下：

```
SQL> archive log list;
Database log mode                Archive Mode
Automatic archival                Enabled
Archive destination              USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence       3670
Next log sequence to archive    3672
Current log sequence             3672
```

脚本 2-15 查看开启是否成功

进程也可以查询到了，我们还发现 arch 进程是允许多进程的，当前是 2 个进程在运行，如下：

```
[oracle@itmapp3 ~]$ ps -ef |grep arc
```

```
oracle 25316 1 0 19:37 ? 00:00:00 ora_arc0_itmtest
oracle 25318 1 0 19:37 ? 00:00:00 ora_arc1_itmtest
oracle 25940 21661 0 19:39 pts/1 00:00:00 grep arc
```

脚本 2-16 查看 Oracle 归档进程

3. 启停的体会

“数据库是如何启动和关闭的，这可以说是数据库最常见的操作了，前面大家已经注意到了在将数据库归档开启时，我是把数据库先关闭，再启动到 mount 状态，然后执行开启归档命令，最后将数据库 Open。

这里我将会结合之前体系结构图来向大家详细介绍一下数据库启动的具体步骤(如图 2-25 所示)，首先看体系结构图的数据库部分，回想起来发现之前我并没有给大家描述参数文件和日志文件：

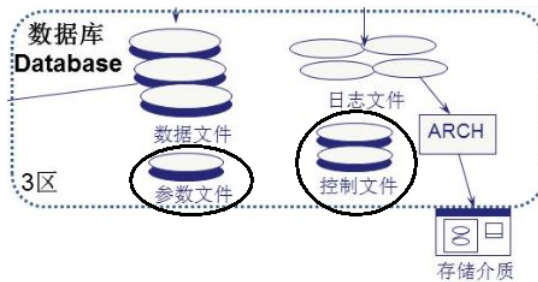


图 2-25 启停的体会

参数文件及控制文件和数据库的启动与关闭是息息相关的，数据库的启动可分为三个阶段，分别是 nomount、mount 和 open。我们在启动的过程中可以直接输入 startup 启动，也可以分成 startup nomount、startup mount 和 alter database open 三步分别启动，下面我们依次描述三个步骤的细节。

① startup nomount 阶段

Oracle 必须读取到数据库的参数文件 (PFILE 或者 SPFILE)，如果读不到该参数文件，数据库根本无法 nomount 成功！如果读到参数文件，将完成一个非常重要的事，就是根据参数文件上的内存分配策略分配相应的内存区域，并启动相应的后台进程，换言之，就是创建了实例 instance。

为了保证数据库可以动态地修改参数，从 Oracle 9i 起，Oracle 引进了 SPFILE 的参数来替代之前仅有单一 PFILE 的一种情况。具体在数据库开启后可以执行如下命令来了解，如下所示就表

收获，不止 Oracle

示是 SPFILE 启动的：

```
SQL> show parameter spfile
NAME                                TYPE                                VALUE
-----                                -                                -
spfile                              string                              /home/oracle/product/10.2.0/db_1/dbs/spfileitmttest.ora
```

脚本 2-17 查看 Oracle 的 spfile 参数情况

一般来说 Oracle 9i 版本以后的数据库是这样一种情况，首先查找 SPFILE 文件，查找不到了再查 INIT.ORA 文件，再查不到，就报错了，nomount 失败。

② startup mount 阶段

实例已经创建了，Oracle 继续根据参数文件上描述的控制文件的名称及位置，去查找控制文件，一旦查找到立即锁定该控制文件。控制文件里记录了数据库中数据文件、日志文件、检查点信息等非常重要的信息，所以 Oracle 成功锁定控制文件，就为后续读取操作这些文件打下了基础，锁定控制文件成功就表示数据库 mount 成功，为实例和数据库之间桥梁的搭建打下了基础。

③ alter database open 阶段

根据控制文件记录的信息，定位到数据库文件、日志文件等，从而正式开通了实例和数据库之间的桥梁。

总结起来，nomount 阶段仅需一个参数文件即可成功，mount 阶段要能够正常读取到控制文件才能成功，而 open 阶段需要保证所有的数据文件和日志文件等需要和控制文件里记录的名称和位置一致，能被锁定访问更新的同时还要保证没有损坏，否则数据库的 open 阶段就不可能成功。

下面我们来演示一下数据库启动的过程，前面大家有见过 startup 的命令，该命令实际是封装了三个阶段的步骤，下面我分步骤执行，让大家体会一下。

```
SQL> startup nomount
ORACLE instance started.
Total System Global Area 2097152000 bytes
Fixed Size                2085192 bytes
Variable Size             520097464 bytes
Database Buffers         1543503872 bytes
Redo Buffers              31465472 bytes
SQL> alter database mount;
Database altered.
SQL> alter database open;
Database altered.
```

脚本 2-18 Oracle 启动三步骤

数据库的关闭过程是启动过程的逆过程，先把数据库关闭，然后数据库和实例之间的

DISMOUNT，最后实例关闭，开辟的内存区消失，后台进程也全部消失。命令就是 `shutdown immediate`，注意这里没有分三阶段执行的命令，都整合在一个 `shutdown immediate` 命令中完成。

```
SQL> shutdown immediate
Database closed.
Database dismounted.
ORACLE instance shut down.
```

脚本 2-19 Oracle 关闭

这时我们再观察发现 Oracle 开辟的 **key=0x66a02988** 且状态 **status** 不为 **dest** 的大小为 **2485125120** 的共享内存段已经消失了：

```
[oracle@itmapp3 ~]$ ipcs -m
----- Shared Memory Segments -----
key          shmid      owner      perms      bytes       nattch     status
0x00020001 0          nmtestv3   666        12          8
0x110280be 1736708    agentv3    666        136         1
0x110280ba 1769477    agentv3    666        30720       1
0x00000000 1081351    oracle     600        393216      2          dest
0x00000000 1114120    oracle     600        393216      2          dest
0x00000000 1146889    oracle     600        393216      2          dest
0x00000000 1179658    oracle     600        393216      2          dest
0x00000000 1212427    oracle     600        196608      2          dest
0x00000000 1245196    oracle     600        393216      2          dest
0x00000000 1277965    oracle     600        393216      2          dest
0x00000000 1310734    oracle     600        196608      2          dest
0x00000000 1343503    oracle     600        393216      2          dest
```

脚本 2-20 观察 Oracle 关闭后的共享内存情况

此外，进程也消失了，我们用 **itsmtest** 的实例名来查找进程，发现不存在。

```
[oracle@itmapp3 ~]$ ps -ef |grep itsmtest
oracle   29666 28673  0 07:34 pts/1    00:00:00 grep itsmtest
```

脚本 2-21 观察 Oracle 进程情况

这里我要让大家加深一下印象，我把数据库再次开启，并只开启到 **nomount** 状态，如下：

```
[oracle@itmapp3 ~]$ sqlplus "/ as sysdba"
Connected to an idle instance.
SQL> startup nomount
ORACLE instance started.
Total System Global Area 2483027968 bytes
```

收获，不止 Oracle

Fixed Size	2086000 bytes
Variable Size	570428304 bytes
Database Buffers	1879048192 bytes
Redo Buffers	31465472 bytes

脚本 2-22 启动 Oracle 到 nomount 状态

这时无无论是共享内存 SGA 还是 Oracle 系列后台进程，都已经出现了：

共享内存段即 ORACLE 的 SGA 立即开辟了

```
[oracle@itmapp3 ~]$ ipcs -m
```

```
----- Shared Memory Segments -----
```

key	shmid	owner	perms	bytes	nattch	status
0x00020001	0	nmtestv3	666	12	8	
0x66a02988	2129921	oracle	660	2485125120	13	

--后面略去

后台进程也立即产生了

```
[oracle@itmapp3 ~]$ ps -ef |grep itmtest
```

oracle	5609	1	0 07:44 ?	00:00:00	ora_pmon_itmtest
oracle	5611	1	0 07:44 ?	00:00:00	ora_psp0_itmtest
oracle	5613	1	0 07:44 ?	00:00:00	ora_mman_itmtest
oracle	5615	1	0 07:44 ?	00:00:00	ora_dbw0_itmtest
oracle	5617	1	0 07:44 ?	00:00:00	ora_lgwr_itmtest
oracle	5619	1	0 07:44 ?	00:00:00	ora_ckpt_itmtest
oracle	5621	1	0 07:44 ?	00:00:00	ora_smon_itmtest
oracle	5623	1	0 07:44 ?	00:00:00	ora_reco_itmtest
oracle	5639	1	0 07:44 ?	00:00:00	ora_cjq0_itmtest
oracle	5642	1	0 07:44 ?	00:00:00	ora_mmon_itmtest
oracle	5644	1	0 07:44 ?	00:00:00	ora_mmln_itmtest
oracle	5646	1	0 07:44 ?	00:00:00	ora_d000_itmtest
oracle	5648	1	0 07:44 ?	00:00:00	ora_s000_itmtest

脚本 2-23 观察 Oracle 启动后内存分配和进程情况

此外，我只要把数据库的参数文件移除或者重命名了，数据库启动时就失败在 nomount 阶段，如果把数据库的控制文件移除或重命名了，数据库启动就会失败在 mount 阶段，如果数据库的任何一个数据文件或者日志文件被移除或者重命名了，数据库就会失败在 open 阶段。这里我就不做试验了，大家如果在自己的电脑搭建过数据库环境，可以自行尝试，大家想不想回去做个试验看看？”

“想！”台下来一致的声音。梁老师虽然已经说得非常清楚了，大家还是很想回去后试验一下加深印象。

“不过切记所有的试验都必须是在你们自己的电脑环境中操作，千万别在生产环境中随便做试验！”梁老师再次强调。

4. 文件的体会

“没有参数文件，实例无法创建，数据库无法 NOMOUNT 成功；没有控制文件，数据库无法 MOUNT；没有数据文件，数据库无法打开使用（此外没有了数据文件，那数据也没地方保存了，数据库也失去意义了）；没有日志和归档文件，数据库就失去了保护伞，变得很不安全了。因此所有的这些文件都非常重要。

我想让大家感受一下 Oracle 数据库中这些参数文件、控制文件、数据文件、日志文件、归档文件存在数据库所在主机的什么位置，又都是通过什么方法查询到的，让大家真真切切感觉到它们的存在，从而进一步加深印象：

参数文件位置：

```
SQL> show parameter spfile;
```

NAME	TYPE	VALUE
spfile	string	/home/oracle/product/10.2.0/db_1/dbs/spfileitmttest.ora

控制文件位置：

```
SQL> show parameter control
```

NAME	TYPE	VALUE
control_file_record_keep_time	integer	7
control_files	string	/home/oracle/oradata/itmttest/control01.ctl, /home/oracle/oradata/itmttest/control02.ctl, /home/oracle/oradata/itmttest/control03.ctl

数据文件位置：

```
[oracle@itmapp3 archive]$ sqlplus "/ as sysdba"
```

```
SQL> select file_name from dba_data_files;
```

FILE_NAME
/home/oracle/oradata/itmttest/users01.dbf
/home/oracle/oradata/itmttest/sysaux01.dbf
/home/oracle/oradata/itmttest/undotbs01.dbf
/home/oracle/oradata/itmttest/system01.dbf
/home/oracle/oradata/itmttest/BOSSWGV3.dbf
/oradata/bosswg_test183_cfg_01.dbf
/oradata/bosswg_test183_cfg_02.dbf
/oradata/bosswg_test183_cfg_03.dbf
/oradata/bosswg_test183_cfg_04.dbf
/oradata/bosswg_test183_cfg_05.dbf

收获，不止 Oracle

```
/oradata/bosswg_test183_perfdata_01.dbf
/oradata/bosswg_test183_perfdata_02.dbf
/oradata/bosswg_test183_perfdata_03.dbf
/oradata/bosswg_test183_perfdata_04.dbf
/oradata/bosswg_test183_techdata_01.dbf
/oradata/bosswg_test183_UNDOTBS_01.dbf
/oradata/TBS_WG_OLD_DEV_1.dbf
17 rows selected.
```

日志文件位置：

```
SQL> select group#,member from v$logfile ;
GROUP#    MEMBER
```

```
-----
      3    /home/oracle/oradata/itmttest/redo03.log
      2    /home/oracle/oradata/itmttest/redo02.log
      1    /home/oracle/oradata/itmttest/redo01.log
```

归档文件位置：

```
SQL> show parameter recovery
```

NAME	TYPE	VALUE
db_recovery_file_dest	string	/home/oracle/flash_recovery_area
db_recovery_file_dest_size	big integer	2G
recovery_parallelism	integer	0

告警日志文件（位于 bdump 目录下，以 alert 打头的文件）：

```
SQL> set linesize 1000
```

```
SQL> show parameter dump
```

NAME	TYPE	VALUE
background_core_dump	string	partial
background_dump_dest	string	/home/oracle/admin/itmttest/bdump
core_dump_dest	string	/home/oracle/admin/itmttest/cdump
max_dump_file_size	string	UNLIMITED
shadow_core_dump	string	partial
user_dump_dest	string	/home/oracle/admin/itmttest/udump

```
[oracle@itmapp3 ~]$ cd /home/oracle/admin/itmttest/bdump
```

```
[oracle@itmapp3 bdump]$ ls -lart alert*
```

```
-rwxr-xr-x 1 oracle  oracle  5305947  Aug 19 08:00  alert_itmttest.log
```

脚本 2-24 查看参数、控制、数据、日志、归档、告警文件

至此，梁老师费尽千辛万苦，终于和大家一起完成了 Oracle 数据库实践的 4 次体会，大家现在还觉得抽象吗？是否觉得面对 Oracle 不再陌生，瞬间亲切了许多？”

“确实是感觉亲切多了！”小莲心中暗自感慨了一番。

结合这堂实践体会课和先前梁老师做的 SQL 脚本执行的系列试验，小莲忽然觉得先前梁老师让自己不断强记的体系结构，现在好像根本不需要去记忆了。甚至觉得，这体系结构图理所当然就应该是这样的。

5. 监听的体会

“梁老师这里补充一个知识点，就是 Oracle 的监听，如果想在远程 A 机器上通过网络访问本地 B 机器上的数据库，B 机器上的数据库必须开启监听。远程的 A 机器只需安装数据库客户端，然后通过读取 A 机器上数据库客户端配置的 TNSNAMES.ORA 的配置文件，即可连接并访问 B 机器的数据库。这里只是简要说明一下，不是课程的重点，详细的文档可以参考 Oracle 官方文档的 CONCEPT 说明。下面我们介绍监听状态的查看，监听的开启，以及监听的关闭。”

以下 **lsnrctl status** 命令是查看监听的状态命令，其中 Listener Parameter File 和 Listener Log File 定位了监听文件 listener.ora 以及对应的日志：

```
[oracle@itmapp3 ~]$ lsnrctl status
LSNRCTL for Linux: Version 10.2.0.4.0 - Production on 21-AUG-2012 14:49:15
Copyright (c) 1991, 2007, Oracle. All rights reserved.
Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=localhost.localdomain)(PORT=21001)))
STATUS of the LISTENER
-----
Alias                     LISTENER
Version                   TNSLSNR for Linux: Version 10.2.0.4.0 - Production
Start Date                17-JUL-2012 07:18:23
Uptime                    35 days 7 hr. 30 min. 52 sec
Trace Level               off
Security                  ON: Local OS Authentication
SNMP                      OFF
Listener Parameter File   /home/oracle/product/10.2.0/db_1/network/admin/listener.ora
Listener Log File         /home/oracle/product/10.2.0/db_1/network/log/listener.log
Listening Endpoints Summary...
  (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=itmapp3)(PORT=21001)))
  (DESCRIPTION=(ADDRESS=(PROTOCOL=ipc)(KEY=EXTPROC)))
Services Summary...
Service "PLSExtProc" has 1 instance(s).
  Instance "PLSExtProc", status UNKNOWN, has 1 handler(s) for this service...
Service "itmpdm" has 2 instance(s).
  Instance "itmpdm", status UNKNOWN, has 1 handler(s) for this service...
  Instance "itmpdm", status READY, has 1 handler(s) for this service...
Service "itmpdmXDB" has 1 instance(s).
  Instance "itmpdm", status READY, has 1 handler(s) for this service...
Service "itmpdm_XPT" has 1 instance(s).
```


收获，不止 Oracle

```
Instance "itmpdm", status READY, has 1 handler(s) for this service...
Service "itmtest" has 1 instance(s).
Instance "itmtest", status UNKNOWN, has 1 handler(s) for this service...
The command completed successfully
```

脚本 2-25 查看监听状态

以下 **lsnrctl stop** 命令是关闭监听的命令：

```
[oracle@itmapp3 ~]$ lsnrctl stop
LSNRCTL for Linux: Version 10.2.0.4.0 - Production on 21-AUG-2012 14:52:46
Copyright (c) 1991, 2007, Oracle. All rights reserved.
Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=localhost.localdomain)(PORT=21001)))
The command completed successfully
```

脚本 2-26 关闭 Oracle 监听

查看发现监听果然被关闭，提示 No listener：

```
[oracle@itmapp3 ~]$ lsnrctl status
LSNRCTL for Linux: Version 10.2.0.4.0 - Production on 21-AUG-2012 14:53:40
Copyright (c) 1991, 2007, Oracle. All rights reserved.
Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=localhost.localdomain)(PORT=21001)))
TNS-12541: TNS:no listener
TNS-12560: TNS:protocol adapter error
TNS-00511: No listener
Linux Error: 111: Connection refused
Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=IPC)(KEY=EXTPROC)))
TNS-12541: TNS:no listener
TNS-12560: TNS:protocol adapter error
TNS-00511: No listener
Linux Error: 2: No such file or directory
```

脚本 2-27 查看关闭 Oracle 监听后的情况

开启命令为 **lsnrctl start**，具体如下：

```
[oracle@itmapp3 ~]$ lsnrctl start
LSNRCTL for Linux: Version 10.2.0.4.0 - Production on 21-AUG-2012 14:54:30
Copyright (c) 1991, 2007, Oracle. All rights reserved.
Starting /home/oracle/product/10.2.0/db_1/bin/tnslsnr: please wait...
TNSLSNR for Linux: Version 10.2.0.4.0 - Production
System parameter file is /home/oracle/product/10.2.0/db_1/network/admin/listener.ora
Log messages written to /home/oracle/product/10.2.0/db_1/network/log/listener.log
Listening on: (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=itmapp3)(PORT=21001)))
Listening on: (DESCRIPTION=(ADDRESS=(PROTOCOL=ipc)(KEY=EXTPROC)))
```

```

Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=localhost.localdomain)(PORT=21001)))
STATUS of the LISTENER
-----
Alias                     LISTENER
Version                   TNSLSNR for Linux: Version 10.2.0.4.0 - Production
Start Date                21-AUG-2012 14:54:30
Uptime                    0 days 0 hr. 0 min. 0 sec
Trace Level               off
Security                  ON: Local OS Authentication
SNMP                      OFF
Listener Parameter File   /home/oracle/product/10.2.0/db_1/network/admin/listener.ora
Listener Log File         /home/oracle/product/10.2.0/db_1/network/log/listener.log
Listening Endpoints Summary...
  (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=itmapp3)(PORT=21001)))
  (DESCRIPTION=(ADDRESS=(PROTOCOL=ipc)(KEY=EXTPROC0)))
Services Summary...
Service "PLSExtProc" has 1 instance(s).
  Instance "PLSExtProc", status UNKNOWN, has 1 handler(s) for this service...
Service "itmpdm" has 1 instance(s).
  Instance "itmpdm", status UNKNOWN, has 1 handler(s) for this service...
Service "itmtest" has 1 instance(s).
  Instance "itmtest", status UNKNOWN, has 1 handler(s) for this service...
The command completed successfully

```

脚本 2-28 开启 Oracle 监听

2.3 体系学习让 SQL 性能提升千倍

2.3.1 一起探索体系学习意义

“同学们，至此老师已经把体系物理结构基本说完了，大家回顾一下，能回忆起我说的内容并全部理解吗？”

“能！”同学们都喊得非常大声。

“确实，梁老师上课如此通俗易懂、生动有趣，让我们理解起来轻松愉快，时间也感觉过得飞快，这真是我听过的最生动的一堂课了。”小莲心中暗自想到，心情有些激动。

“不过在学习体系结构的课堂里，梁老师认为最重要的内容还没给大家说，那就是学习体系结构的意义。

梁老师上过的课很多，做过的项目更是数不胜数，因此有条件见识不少的新人。这些新人少

收获，不止 Oracle

数成长迅速，短时间内就成为部门甚至公司的技术骨干，而大多数人虽然同样努力却始终进步缓慢，难以独当一面。

这是为什么呢？

主要因为不少人为了学习知识而学习，从不曾想过这些知识学习的意义；还有不少讲师为了授课而授课，却很少提及这些授课内容对工作能带来哪些帮助；最后，新人都能理解大多数知识点，但是由于缺少这些想法和意识，所以一直无法突破提升。

下面老师开始给大家举例说明，我为什么要花这么大精力给同学们描述体系结构的课程，为了什么？

请大家竖起耳朵听老师说一句话：‘Oracle 的体系结构运行机制不就是那样吗？我们学习了这些原理，又不能改变数据库的机制，它该怎么运行就怎么运行，学习有啥意义？还不如实际点跳过这个体系结构学习，直接教我们如何建数据库、建用户、迁移数据、写 SQL 等工作中会常遇到的操作，别尽整一些没用的东西。’

在座的有 40 位同学，请大家每 10 人形成一个小组开始针对这句话进行 10 分钟的探讨，并派代表上来回答。

另外我还可以说得更简单点，就是让你们去思考知道这些体系知识的你和那些不知道这些知识的人，在操作数据库时会有差异吗？你有什么优势？好好讨论。”

2.3.1.1 同学们不知所何用

台下讨论得很热烈，很快 10 分钟过去了，敬昱作为代表首先上场讲述自己小组的观点了。

“梁老师，我们讨论的结果是您说的这些没啥用，相比不懂这些知识的人没啥优势。”

话音刚落，台下笑成一片，看来梁老师足够随和，同学们也就足够大胆了。

“您说 Oracle 的共享池是为了第一次执行时保存解析过程，避免第二次执行再做代价高昂的解析。而数据缓存区是为了第一次获取数据时将这些数据从磁盘读到数据缓存区，以便第二次可以避开磁盘查找，直接从数据缓存区找到数据，从而避免物理读。这些东西您描述得那么形象，我们自然都能理解，可是现实中我们用数据库就是了，关心这些也没意义啊，不理解这些的人和俺们不是一样都在操作数据库，不懂这个的人也照样查询数据库，更新数据库，差别就是我们可能细心点，留意到好像相同的语句第二次会更快一点，并且知道原因，仅此而已啊。”

接下来轮到下一组同学代表了，扎马尾辫的晶晶上台来了，她和敬昱所在小组持同样的观点，不过这个细心的女孩还提到 PGA 区的排序，她说道：“梁老师您说排序在 PGA 内存区中操作，如果尺寸装得下就在内存中完成，否则超出部分会在临时表空间中完成排序，造成性能低下。我觉得知道这点固然好，但是意义不大，在特定排序需求的场合下，我们难道就不排序吗？知道这个和不知道这些知识点的同学都要写 order by 等关键字，没啥差别啊。”

“还有您说的关于日志切换，”晶晶继续说，“LGWR 从日志缓冲区中把日志写出，并写进日志文件，写满第 1 个写第 2 个……全部写完要循环写的时候，出现覆盖日志时要把即将被覆盖的

日志先转移出去，形成归档文件保存起来，从而用于数据库的备份与恢复。我觉得我知道这个很开心，不过实际上，不知道这些的人，他只要学会怎么操作这些恢复文件，用于恢复就可以了啊。我现在还只是知道日志可以恢复的原理是什么，却不知道如何恢复，不知道命令是什么，我感觉和不懂这些原理的人差别不大啊，直接告诉我恢复命令岂不是更实在？”

紧接着林君做代表上来了，这组同学还是持同样的看法，对自己学这些的意义有些茫然。“梁老师，我们很认真地听明白了一致读的原理，您举的转账的例子太生动了，我们不明白都不行了。但是我觉得也没啥实用性，比如我查询很久后返回了一些可能期间被频繁更新的数据，我知道其中可能涉及一致读，但是不知道一致读的人也不影响他返回结果啊，差别就是我们心里比他明白有可能产生了一致读，并且原理是这样这样的，但是工作中没感觉到有看得见的优势啊。”

小莲有些不好意思地代表第四组上台了，她虽然觉得梁老师上课上得非常生动，但是梁老师非要自己说懂这些体系结构原理和完全不懂体系结构的人在操作数据库上优势有哪些，还真是想不出来。

第四小组讨论后最终是问出了这个问题：“梁老师，共享池的解析是开销很大的操作，我对其中的分析获取最优执行计划印象很深刻，您说 Oracle 到底是选择索引读还是全表扫描，是由 Oracle 自己来决定的，判断代价谁高谁低，代价低的胜出。如此我们何必要关心这个细节，我们又不能左右他们代价的算法，不是听天由命？仅此而言，我就觉得和不懂这个知识细节的同学没差别啊。”

“批判大会啊，你们让老师觉得自己罪孽深重，需要好好忏悔忏悔了。”

等下老师会去公司财务部一趟，请求财务把我今天的工钱扣回去，而且浪费了大家这么长时间了，应该再恳请公司罚半个月工资，以此表示自己对大家的歉意，你们看这样好不好，可以原谅梁老师了吗？”

梁老师的调侃让台下笑得前俯后仰。

“梁老师的葫芦里肯定要卖什么猛药了！”笑归笑，小莲早已打起十二分精神，准备仔细听梁老师是怎么说的。

2.3.1.2 实际上大有用武之地

“同学们积极参与、善于思考，这是非常棒的，你们是我见过的最认真、最活跃的一批学生了。”梁老师首先表扬了一下同学们。

“老师本来上得好好的，很快就可以结束今天的课程了，忽然自己没事找事引火烧身，引发你们产生对了解数据库体系结构有无意义的疑问。现在问题抛出来了，老师要是不给你们一个交代继续上新的课程，估计会被你们赶下讲台。”

那老师来依次回答大家的问题，探讨学习的意义。

先回答敬昱的提问。我举个例子，假如某数据库是一个很大的数据库，数据量庞大，访问量非常高，而共享池却非常小，那会怎么样？肯定共享池很快就被撑满了，缓存的东西要不断地被

收获，不止 Oracle

挤出，结果很多 SQL 都难以避免硬解析，因为很快被挤出共享池消失得无影无踪了，于是整个数据库开始运行缓慢。

同学们，此时我们要做的就是……”

“加大共享池！”

“如果是自动管理模式，就是加大 SGA 的大小！”

台下回答得很迅速。

“难道这不是了解体系结构的好处吗？”

台下有不少人点头了。

“再比如说，某主机总共才 4G 的内存，而运行在其平台上的数据库是一个几乎没有什么访问量的小数据库，可能 100M 的共享池就足够了，却被开辟了 3G 的 SGA 内存，500M 的 PGA 内存。但是由于操作系统内存不足，导致主机运行缓慢，从而导致数据库运行缓慢，我们要做的是……”

“减少 SGA 大小！”

“那如果由于数据缓存区过小导致大数据量的数据库产生大量的物理读，怎么办？”

“梁老师，这和共享池情况是类似的，SGA 自动管理的情况下，加大 SGA 的大小，也等同于加大了数据缓存区的大小，这样数据缓存区够大，装的就多，物理读自然就减少了，性能自然就提高了。”敬昱起身回答这个问题。

“敬昱，你们现在还觉得了解体系结构没意义吗？一点都不懂体系结构的人，你觉得他该从哪里入手解决这个问题呢？”

敬昱有些不好意思地笑了笑。

“那我们来看看晶晶的问题，晶晶说的是了解排序在 PGA 区没有多大意义，如果一个尺寸很大的排序由于内存无法装下要在磁盘中进行，而操作系统却闲置着大量的内存未使用，我们要做的就是……”

“增加 PGA 大小，争取容纳下排序的尺寸，从而避免物理排序。”晶晶听到梁老师故意放慢语气后，起身接下梁老师的话，回答了。

“那如果主机的内存不足，而某特定数据库几乎没有什么排序的应用，我们就可以……”梁老师又放慢了语调。

“减小 PGA，腾出占用的内存分配给操作系统使用。”还是晶晶回答了。

“假如一个数据库系统存在大量的更新操作，产生了大量的日志需要从 REDO BUFFER 中写出到日志文件里，结果梁老师说过的那种日志写满然后切换到下一个日志的频率不断加快。这里大家要注意一点，切换过程中数据库需要等待切换完成才可以正常运作，因为切换没完成，LGWR 就无法把 REDO BUFFER 的数据继续写出来，而数据库中 REDO BUFFER 产生的记录总是先于数据缓存区产生的，这是串行的顺序，那此时数据库更新的动作就根本不可能成功，一定要等待日志切换成功，世界才恢复正常次序。

那现在日志频繁地切换，我们的更新一直在停停走走的，我们要做的事是……”

整整过了 30 秒，晶晶终于明白过来了：“梁老师，日志组的这些日志越大，就越经得住写，切换就越慢，等待就越小，所以加大这些日志文件的尺寸，是这样吗？”

“晶晶，你还觉得了解体系结构没优势吗？不懂体系结构，你能这样考虑问题入手优化系统吗？”

晶晶也不好意思地笑了笑。

“接下来是林君的问题，他们组提到查询的一致性问题的，认为这个细节无须掌握，那你们仔细回想一下，什么时候这个查询会出错？”

“当前镜像无法找回原来的记录，被覆盖重写时就会报错退出，好像错误号是 ORA-01555，Oracle 宁愿查询失败也不愿意查询出错误不一致的结果。”林君回答得很流利。

“记得很清楚嘛，看来林君对梁老师账户凭空多出 500 元没请客而耿耿于怀啊。”

又是一阵大笑。

“现在某应用系统因为查询老出 ORA-01555 错误返回不了结果给下一个模块使用，导致生产出现故障了，你们来帮他们出主意吧，我们能出的建议是……”

“检查这个语句为什么执行这么慢，让这个语句执行得快一点，就不容易被更新直至覆盖重写啊。”林君稍作思考就回答了。

“对啊，这不是回答得非常好吗？优化也许是加个索引，也许是清理表的历史数据，让表的记录小一些。但是不管是什么方法，语句执行快了，这个失败的概率就大大降低了，仔细想想梁老师之前上课内容，还有没有其他方法也可能可以解决这个查询失败问题？”

“梁老师，增大 `undo_retention` 的取值，比如写一个很大的时间，让回滚段在这段时间内都不允许被覆盖重写。”小莲起身回答了。

“好了，现在林君这个小组的同学还认为学习体系结构没意义吗？”

现在不止是林君不好意思地笑了笑，所有同学都觉得有些难为情了。

“不过这里要注意一点，`undo_retention` 只是参考保留时间并非强制，除非另外设置 UNDO 表空间的 `Retention Guarantee` 属性，使之强制保留。此外也可考虑增加 UNDO 表空间的大小。现在轮到最后一组同学的问题了，小莲提出的是 Oracle 共享池解析过程中，选择全表扫还是索引读是我们无法左右，所以知道这个也没意义。确实，有时索引高效有时全表扫描高效，这和返回的记录数多少很有关系。返回绝大部分的数据，一般情况下倾向于全表扫，返回少量的数据，一般情况下倾向于索引读。

我现在想问的是，你如果没建索引，会有索引扫描吗？”

“当然不会！”小莲不假思索地回答。

“那还没意义吗，了解体系结构的人知道优化器能解析出代价最低的执行计划，那你就别让优化器巧妇难为无米之炊，可以多送点武器供他比较选择，看看哪个兵器称手啊，不懂体系结构的人可以想到这一层吗？当然索引本身也是一个非常重要的知识，后续我应该还有机会给大家做系列培训，到时候我会花费很长时间给大家详细描述索引的。”

“哎呀，丢人了，我问的问题怎么这么傻啊？”小莲忽然有一种从梦中惊醒过来的感觉，顿

收获，不止 Oracle

时脸红到耳根。

“我们怎么都不会往梁老师回答的这些方面去想问题呢？”台下有些寂静，大家都不约而同地想到了这个问题。

2.3.2 单车到飞船的经典之旅

“刚才老师回答了不少同学的疑问，说明了学习体系结构还是有意义的。看来只要善于思考，善于联系，我们就能很容易地应用所学的知识解决遇到的问题。

现在我给大家讲一个具体的案例，通过执行系列等价 SQL 语句的性能差异，来分析其中的原因所在。

2.3.2.1 未优化前，单车速度

首先构造环境，保证下列语句执行过了，t 表已经存在：

```
sqlplus ljb/ljb
drop table t purge;
create table t ( x int );
--将共享池清空
alter system flush shared_pool;
```

脚本 2-29 单车到飞船试验前的准备工作

有一个开发人员写了类似如下的一个简单存储过程，实现了将 1 到 10 万的值插入 t 表的需求，具体如下：

```
create or replace procedure proc1
as
begin
    for i in 1 .. 100000
    loop
        execute immediate
            'insert into t values ( '||i||' )';
        commit;
    end loop;
end;
/
----这里要记得先预先执行一遍，将过程创建起来！
```

脚本 2-30 单车到飞船试验前构造 proc1

该语句从需求实现上没有任何问题，我们执行如下：

```
SQL> connect ljb/ljb
```

```

已连接。
SQL> drop table t purge;
表已删除。
SQL> create table t ( x int );
表已创建。
SQL> alter system flush shared_pool;
系统已更改。
SQL> set timing on
SQL> exec proc1 ;
PL/SQL 过程已成功完成。
已用时间: 00: 00: 42.87
SQL> select count(*) from t;
COUNT(*)
-----
100000

```

脚本 2-31 首次试验 42 秒完成，仅是单车速度

该语句用 42 秒时间完成 10 万条记录的插入，大家觉得速度快吗？”

“1 秒钟插入 2 千多条记录，速度好快啊！”小莲在数学上的反应还是很敏捷。

“哦，你希望还能更快一点吗？”梁老师问。

“哦，当然希望了，肯定是越快越好了。”小莲回答得不假思索。

“其实这个简单的过程如果想更快，靠的就是对体系结构的理解，下面大家跟我查询一下该过程执行中，数据库共享池中的相关情况。

共享池中缓存下来的 SQL 语句以及 HASH 出来的唯一值，都可以在 v\$sql 中对应的 SQL_TEXT 和 SQL_ID 字段中查询到，而解析的次数和执行的次数分别可以从 PARSE_CALL 和 EXECUTIONS 字段中获取。

由于这个过程 PROC1 执行的是 insert into t 的系列插入，于是我们执行如下语句来查询 PROC1 在数据库共享池中执行的情况，具体如下：

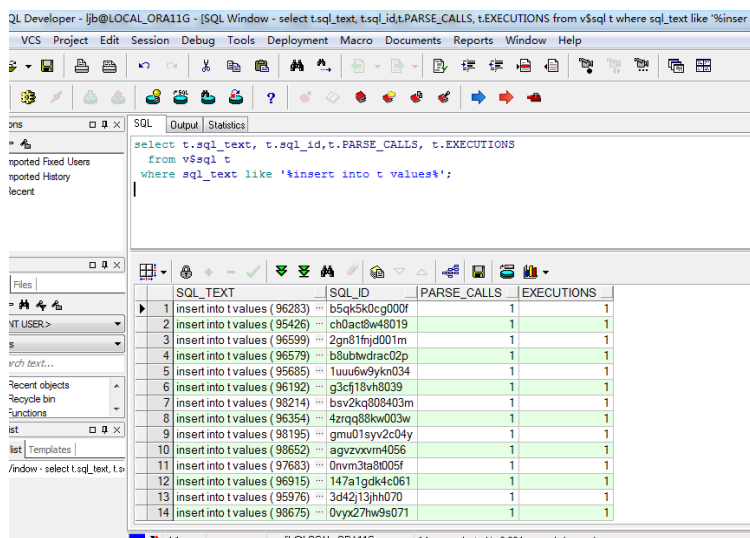
```

select t.sql_text, t.sql_id, t.PARSE_CALLS, t.EXECUTIONS
from v$sql t
where sql_text like '%insert into t values%';

```

脚本 2-32 原来是因为未用绑定变量

为了让 SQL 语句及展现结果看起来更美观，我将上述 SQL 在 PL/SQL DEVELOPER 工具中查询（这个工具有着很友好的界面，尤其适合数据库开发人员使用，因此除了介绍先前的 sqlplus 工具外，这里也顺便提一下这个 PL/SQL DEVELOPER 工具），发现共享池中有大量的类似 SQL 语句，而 SQL_ID 各自不同，每个语句都只是解析 1 次，执行 1 次，解析了 10 万次了，怪不得速度如此之慢，如图 2-26 所示。



	SQL_TEXT	SQL_ID	PARSE_CALLS	EXECUTIONS
1	insert into t values (96283) ---	b5qk5k0cg000f	1	1
2	insert into t values (95426) ---	ch0ac18w48019	1	1
3	insert into t values (96599) ---	2gn81fjd001m	1	1
4	insert into t values (96579) ---	b8ubtwdrac02p	1	1
5	insert into t values (95685) ---	1uuu6w9ykn034	1	1
6	insert into t values (96192) ---	g3cfj18vh8039	1	1
7	insert into t values (98214) ---	bsv2kq808403m	1	1
8	insert into t values (96354) ---	4zrqg88kw003w	1	1
9	insert into t values (98195) ---	gmu01syv2c04y	1	1
10	insert into t values (98652) ---	agvzvxxvm4056	1	1
11	insert into t values (97683) ---	0nm3ta8005f	1	1
12	insert into t values (96915) ---	147a1gdk4c061	1	1
13	insert into t values (95976) ---	3d42j13jh070	1	1
14	insert into t values (98675) ---	0vpx27hw9s071	1	1

图 2-26 绑定变量

2.3.2.2 绑定变量，摩托速度

此时我们这么想，要是 insert into t values (99898)、insert into t values (99762) 等这 10 万条语句如果都能合并成一种写法，比如用变量代替具体值，成为 insert into t values (:X) ，那岂不是这 10 万条语句可以被 HASH 成一个 SQL_ID 值，不就可以做到解析 1 次，执行 10 万次了？这就大大减少了解析时间。

这就是数据库的一个典型优化，绑定变量优化！

接下来我们将 proc1 改进为 proc2，具体写法如下：

```
create or replace procedure proc2
as
begin
    for i in 1 .. 100000
    loop
        execute immediate
            'insert into t values (:x)' using i;
    commit;
    end loop;
end;
/
```

----这里要记得先预先执行一遍，将过程创建起来！

脚本 2-33 第 2 次改进，将 proc1 改造成有绑定变量的 proc2

接下来我们继续执行测试 `proc2` 过程（注意表重建的目的是为了公平，测试都在无记录的空表上进行，并且共享池都清空）：

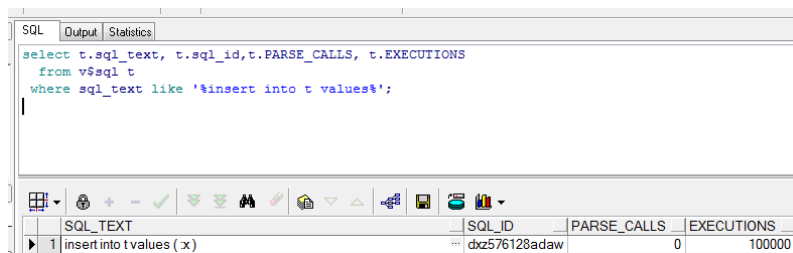
```
SQL> drop table t purge;
表已删除。
SQL> create table t ( x int );
表已创建。
SQL> alter system flush shared_pool;
系统已更改。
SQL> set timing on
已用时间: 00: 00: 00.00
SQL> exec proc2;
PL/SQL 过程已成功完成。
已用时间: 00: 00: 08.41
SQL> select count(*) from t;
COUNT(*)
-----
100000
```

脚本 2-34 第 2 次改进后 8 秒完成，单车变摩托

这次我们惊奇地发现，速度从原来的 42.87 秒缩减为 8.41 秒，大幅度提升了，每秒插入记录数达到 1 万多条，大大超过之前的每秒插入 2 千多条的速度，为什么会这么神奇呢？”梁老师停下来问大家。

“因为语句被绑定变量了，解析次数变少了！”梁老师的话同学们记得很牢。

“很好，我们一起看看吧，看看有啥变化（如图 2-27 所示）：



SQL_TEXT	SQL_ID	PARSE_CALLS	EXECUTIONS
1 insert into t values (x)	dxz576128adaw	0	100000

图 2-27 解析与执行次数

虽然插入的语句值各不相同，但是都被绑定为 `x`，所以被 HASH 成唯一一个 HASH 值，名称为 `dxz576128adaw`，很明显可以看出解析 1 次，执行 10 万次，这就是速度大幅度提升的原因了。这下大家对这个速度满意了吧。”

小莲这下才发现，原来简单的体系结构后面还真有不少玄机啊，也不只是上堂课梁老师说的

收获，不止 Oracle

加大减少共享池这么简单，小莲觉得今天的课太值了。

2.3.2.3 静态改写，汽车速度

“大家还想不想再快一点？”梁老师这次开口吓了大家一跳。

“梁老师您太贪心了吧！”晶晶打趣地说道，看来课堂上同学们还真是被梁老师带动得无拘无束了，什么话都敢说啊。

“其实真的是还能再快的，你们看看这两个过程，是否觉得哪里写得有点别扭？”梁老师提醒大家认真看 `proc1` 和 `proc2`。

“梁老师，这个 `execute immediate` 和双引号是啥意思啊，为什么不直接写成 `insert into t values (i)` 啊？”刚才开玩笑的曾祥也提问了。

“不错！终于看出来了，`execute immediate` 是一种动态 SQL 的写法，常用于表名字段名是变量、入参的情况，由于表名都不知道，所以当然不能直接写 SQL 语句了，所以要靠动态 SQL 语句根据传入的表名参数，来拼成一条 SQL 语句，由 `execute immediate` 调用执行。但是这里显然不需要多此一举，因为 `insert into t values (i)` 完全可以满足需求，表名就是 `t` 啊。

我们来改写成 `proc3`，如下：

```
create or replace procedure proc3
as
begin
    for i in 1 .. 100000
    loop
        insert into t values (i);
    commit;
    end loop;
end;
/
```

----这里要记得先预先执行一遍，将过程创建起来！

脚本 2-35 第 3 次改进，将 `proc2` 改造成静态 SQL 的 `proc3`

接下来我们继续测试 `proc3` 的性能，也是在公平的环境下操作的，如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t ( x int );
表已创建。
SQL> alter system flush shared_pool;
系统已更改。
SQL> set timing on
SQL> exec proc3;
```

PL/SQL 过程已成功完成。

已用时间: 00: 00: 06.25

SQL> select count(*) from t;

COUNT(*)

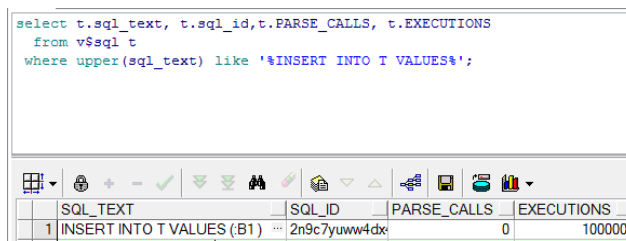
100000

脚本 2-36 第 3 次改进后 6 秒完成，摩托变汽车

大家看看，现在是什么情况？” 梁老师笑着说。

“哇，又快了！” 晶晶惊叫起来，引来了一片笑声。

“为什么会快了，我们分析分析，这个语句肯定有用到绑定变量，一般来说，静态 SQL 会自动使用绑定变量，我们来查看查看（如图 2-28 所示）。



The screenshot shows a SQL query in the editor and its execution statistics in a table below. The query is: `select t.sql_text, t.sql_id, t.PARSE_CALLS, t.EXECUTIONS from v$sql t where upper(sql_text) like '%INSERT INTO T VALUES%';`

	SQL_TEXT	SQL_ID	PARSE_CALLS	EXECUTIONS
1	INSERT INTO T VALUES (B1) ...	2n9c7yuuw4dx	0	100000

图 2-28 静态 SQL 解析与执行次数

果然如此，proc3 也实现了绑定变量，而且动态 SQL 的特点是执行过程中再解析，而静态 SQL 的特点是编译的过程就解析好了。这点差别就是速度再度提升的原因。现在大家对这个速度满意了吧。”

台下纷纷点头。

2.3.2.4 批量提交，动车速度

“还能再快一点吗？” 梁老师再次发问引发台下一片大笑，因为大家这时都认为老师是在开玩笑。

“别笑，我可不是开玩笑哦，同学们再执行这个 proc3，看看有啥新发现。”

“梁老师，我看到 commit 了，我觉得应该放在循环的外面而不是里面。放在里面意味着每插入 1 条，就要提交 1 次，那放在循环里就要提交 10 万次，而放在循环外就是全部插入完后提交 1 次，我觉得少提交更好。” 细心的小莲发现了 commit 位置的不同。

“说得非常好，commit 触发 LGWR 将 REDO BUFFER 写出到 REDO BUFFER 中，并且将回滚段的活动事务标记为不活动，同时让回滚段中记录对应前镜像记录的所在位置标记为可以重写，切记 commit 可不是写数据的动作哦，写数据将数据从 DATA BUFFER 刷出磁盘是由 CKPT

收获，不止 Oracle

决定的，前面大家应该有印象。

所以 `commit` 做的事情开销并不大，单次提交可能需要 0.001 秒即可完成。另外不管多大批量操作后的提交，仅针对 `commit` 而言，也是做这三件事，所花费的总时间不可能超过 1 秒。打个比方，批量 100 万条更新执行后完成 `commit` 的提交可能也就需要 0.8 秒，但是 $100\text{万} \times 0.001$ 的时间可是远大于 1×0.8 的时间了。

下面我们来做个试验，将 `proc3` 改写为 `proc4`，具体如下：

```
create or replace procedure proc4
as
begin
    for i in 1 .. 100000
    loop
        insert into t values (i);
    end loop;
    commit;
end;
/
---这里要记得先预先执行一遍，将过程创建起来！
```

脚本 2-37 第 4 次改进，将 `proc3` 改造成提交在循环外的 `proc4`

接下来我们继续测试 `proc4` 的性能，也是在公平的环境下操作的，如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t ( x int );
表已创建。
SQL> alter system flush shared_pool;
系统已更改。
SQL> set timing on
SQL> exec proc4;
PL/SQL 过程已成功完成。
已用时间: 00: 00: 02.18
SQL> select count(*) from t;
COUNT(*)
-----
100000
```

脚本 2-38 第 4 次改进后 2 秒完成，汽车变动车

速度是多少？”梁老师回头看着大家。

“哇，2 秒就完成了！”还是晶晶的惊叫声。

“等同于每秒5万条记录，原先可是每秒2千多条啊，不可思议！”小莲数学换算总是最快。

2.3.2.5 集合写法，飞机速度

“还能更快吗？”梁老师没完没了啊。

“不可能！”这下同学肆无忌惮地大笑了，他们知道梁老师这回肯定是故意的了。

“不可能？看来你们又认为老师在开玩笑，如果老师能让插入该表的动作更快，你们请老师吃饭，否则老师请你们吃饭？”

“没问题！”同学们应答得好干脆啊，他们其实心里也知道，这请客的说法一定是开玩笑，课堂的气氛被调到了最高潮了。

“好吧，同学们，大家看这条SQL语句的写法，如下：

```
insert into t select rownum from dual connect by level<=100000;
```

表示我从1到10万依次插入到t表中，完全满足刚才的需求，不过这种写法大家可能略感陌生，结果却是对的。现在我们执行如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t ( x int );
表已创建。
SQL> alter system flush shared_pool;
系统已更改。
SQL> set timing on
SQL> insert into t select rownum from dual connect by level<=100000;
已创建 100000 行。
已用时间: 00: 00: 00.25
SQL> commit;
提交完成。
已用时间: 00: 00: 00.00
SQL> select count(*) from t;
COUNT(*)
-----
100000
```

脚本 2-39 第5次用集合写法后0.22秒完成，动车变飞机

大家认真看看完成的时间吧。”梁老师得意地回过头望着大家。

“0.25秒！”这次不再是晶晶一人的声音了，是全部同学一起喊起来了。

“每秒插入40万条，太快了吧！”小莲再次喊出声来。

“为什么会快这么多呢？其实是因为原先的过程变成了SQL，一条一条插入的语句变成了一

收获，不止 Oracle

个集合的概念，变成了一整批地写进 DATA BUFFER 区里，好比你要运砖头到目的地，一种是一块砖头拿到目的地，再返回拿第二块，直到拿完全部。而另一种是全部放在板车一起推至目的地，只是这里的目的地是 DATA BUFFER 区而已。

听明白了吗？”

“听明白了！”大家都很激动，梁老师真像一个魔术师。

“听明白就好，现在你们该好好想想请老师晚上去哪里吃饭了。”

同学们都乐了，大家心里都知道梁老师是在开玩笑。

2.3.2.6 直接路径，火箭速度

“没声音啊，看来大家怕花钱啊，要不老师再给同学们一个机会，如果我还能让这个插入语句更快，大家就请客，否则老师请客，如何？”

梁老师有完没完啊，都每秒钟 40 万条的速度了，还想快，疯了吧。同学们议论纷纷。

“那我开始了，因为前面完成时间都已经到零点几秒了，太小会有误差，所以我准备把数据量放大 100 倍，10 万条改为插入 1000 万条，前面飞机速度的语句变成如下：

```
SQL> connect ljb/ljb
已连接。
SQL> drop table t purge;
表已删除。
SQL> create table t ( x int );
表已创建。
SQL> alter system flush shared_pool;
系统已更改。
SQL> set timing on
SQL> insert into t select rownum from dual connect by level<=10000000;
已创建 10000000 行。
已用时间: 00: 00: 23.22
SQL> commit;
```

脚本 2-40 试验准备，将集合写法的试验数据量放大 100 倍

发现插入 1000 万条记录完成的时间是 23 秒多，大致为每秒钟 43 万条记录，和插入 10 万条记录时的速度每秒 40 万条大体接近，下面我们改用 create table 的直接路径方式来新建 t 表，看看这样的方法速度能否有提升。

```
SQL> drop table t purge;
表已删除。
已用时间: 00: 00: 11.07
SQL> alter system flush shared_pool;
系统已更改。
```

```
已用时间: 00:00:00.02
```

```
SQL> set timing on
```

```
SQL> create table t as select rownum x from dual connect by level<=10000000;
```

```
表已创建。
```

```
已用时间: 00:00:10.14
```

脚本 2-41 第 6 次改进，直接路径让飞机变火箭

测试结果是，速度又有了 2 倍多的提升，只需要 10 秒即可完成，等同于插入速度为每秒钟 100 万条，要不是亲眼所见，还真不敢相信吧。

同学们知道这是为什么吗？真正的原因在于，insert into t select的方式是将数据先写到 DATA BUFFER 中，然后再刷到磁盘中。而 create table t 的方式却是跳过了数据缓存区，直接写进磁盘中，这种方式又称之为直接路径读写方式，因为原本是数据先到内存，再到磁盘，更改为直接到磁盘，少了一个步骤，因而速度提升了许多。

直接路径读写方式的缺点在于由于数据不经过数据缓存区，所以在数据缓存区中一定读不到这些数据，因此一定会有物理读。但是在很多时候，尤其是海量数据需要迁移插入时，快速插入才是真正的第一目的，该表一般记录巨大，DATA BUFFER 甚至还装不下其十分之一、百分之一，这些共享的数据意义也不大，这时，我们一般会选择直接路径读写的方式来完成海量数据的插入。

同学们，听明白了没，现在大家甘心请客了吧？”梁老师时刻不忘调侃。

台下都听呆住了，梁老师这节课真是神了，大家半天没回过神来。

2.3.2.7 并行设置，飞船速度

“大家这次请客是请定了，具体时间老师通知哦。”

台下哈哈大笑，大家还是回过神来了，今天这节课给梁老师弄得一愣一愣的。

“同学们，插入语句还能再快吗？”

这下台下同学们都不敢应答了，梁老师没完没了，却每次都能不断加速，而且用到的知识又和体系知识息息相关，让自己一听就明白了。可以看出梁老师这个经典案例的例子是精心构造的，真可以说是经典中的经典。

“看来是被我给忽悠住了，大家都不敢应了。”梁老师笑着说道，“其实遇到性能好的机器，还是可以大幅度提升性能的，大家看如下语句，我设置日志关闭 nologging，并且设置 parallel 16 表示用到机器的 16 个 CPU，结果在笔记本环境收效不是很明显，因为我的环境是单核的机器。

后来我把如下 SQL 运行在强劲的服務器上，有 16 个 CPU，下面的语句仅仅在 4 秒不到的时间内就完成了，速度相对于前面的火箭速度而言，快多了，几乎是每秒钟 300 万条的插入速度，具体如下：

```
drop table t purge;
```

```
alter system flush shared_pool;
```



```
set timing on
create table t nologging parallel 64
as select rownum x from dual connect by level<=10000000;
```

脚本 2-42 第 7 次改进，并行原理让火箭变飞船

不过并行最大的特点就是占用了大多数 CPU 的资源，如果是一个并发环境，很多应用在跑，因为这个影响了别的应用，导致别的应用资源不足，将引发很多严重问题，所以需要三思而后行，了解清楚该机器是否允许你这样占用全部的资源。

好了，今天我以一条最简单的插入顺序数字进某表的 SQL 语句为例，方法从存储过程实现到仅用单条 SQL 完成；速度从原先的每秒 2 千多条提升到每秒 300 万条，这中间都做了哪些改进呢，是否都和体系结构有关呢？

希望大家好好复习这一节的经典案例，回忆一下速度的 6 次飞跃之旅，好好感受一下灵活应用知识的无限魅力。”

2.3.3 精彩的总结与课程展望

“今天的课程即将结束，大家用心地思考讨论 15 分钟，想想今天一天都收获了什么，学到了什么，老师会提问大家一些问题，让大家回答，希望各位能积极参与、无拘无束、畅所欲言。”

时间确实过得太快了，从早上到现在仅课堂时间就已经 7 个半小时了，大家居然丝毫不感受不到上了一整天课的疲倦，都为沉浸在精彩纷呈、跌宕起伏的课程中而兴奋不已。小莲闭上眼睛，开始从头到尾回忆并思考今天的上课内容，其他同学也有和小莲一样闭目思考的；也有三五成群相互讨论的；也有独自翻阅自己记录的笔记的；还有打开电脑学习梁老师提供的 PPT 教材的……

“老师今天上了什么内容？”

“体系结构，说得很详细，还有不少生动的例子。” 敬昱喊声最大，其他声音的内容也大致如此。

“那最大的收获有哪些呢？” 梁老师继续提出第 2 个问题。

“明白了体系物理结构的原理。”

“知道体系结构学习的意义。”

“听到了很多案例，以后可以借鉴。”

台下七嘴八舌……

2.3.3.1 最大的收获应该是思想

“同学们，你们虽然回答得还不错，但是梁老师并不是很满意，你们知道原因吗？”说到这里梁老师停顿下来看着大家。

台下顿时安静下来，等着梁老师继续往下说。

首先，最遗憾的是，我在问今天上什么内容时，没听到有人说我早上总结的**学习方法、学习路线**这部分内容，而这部分内容，我觉得，重要程度甚至超过了接下来的体系结构的描述。

我谈手机，是要让大家知道学习首先需要学会分类总结。

我描述了角色和具体知识点，让大家知道学习首先要学会把握先后顺序，抓住重点。

我说的知识获取的途径，是要让大家知道自学其实是非常重要的。虽然梁老师后续还会继续给大家上系列课程……”

啪啪啪，台下又突然响起了掌声，打断了梁老师的话。

梁老师一愣，笑笑继续往下说，“但是梁老师永远不会教你们建表建索引以及写具体 SQL 的语法和命令，这需要你们自己去学习，否则这就不是工作技能培训，而是大学甚至中学的课堂学习了。我只会教大家最重点的知识、思想及案例，绝不是所有的知识点，语法更不在上课的范畴内，所以我给大家提到具体获取知识的途径，其中官方文档的 CONCEPT 务必仔细阅读一遍，内容不多，但是说明了概念，具体的知识细节提供了详细的链接。

那还需要老师来上课吗？当然需要，因为我们要快速进步，而不是缓慢前进。官方文档虽然全面、正确，但是永远不会教我们什么时候选择什么技术，什么时候不能选择什么技术；也不会教我们如何将系列技术巧妙结合；不会和我们分享经典的案例；更不会教我们解决问题的思想和方法论。而这些才是最重要的，就是梁老师要教给大家的，但是前提是必须先把官方文档的 CONCEPT 熟读完毕，能很快查阅到某些技术的具体语法及步骤，比如建表建索引、如何写一个查询语句。记住，梁老师不会去教这些可以简单搜索到的，比较死板的东西，没有必要也没有时间，这方面要靠自己，方法已经告诉你们了。”

说到这里，梁老师停下来，对大家说，“老师是不是有点遗憾啊？”

同学们都不好意思地笑了笑，小莲也觉得感触颇深，梁老师可谓用心良苦，其中那个描述什么角色学哪些知识点的 Excel 图估计也整得煞费苦心，自己怎么就没好好体会呢。

“老师还有遗憾，你们最大的收获提及明白了体系结构的原理，这没错，将来会对数据库相关工作带来很大的好处。还有同学说知道了了解体系结构的意义，也挺好的。不过如果有人说**借鉴到了 Oracle 设计的思想**，我将更满意。

这里的共享池和数据缓存区显然是遵循了一个在少做事甚至不做事的情况下，完成相同的工作任务的原理，这不是一个非常完美的优化思想吗？后续的课程中，如何优化是其中的一个重点，大家会发现我反复提及这个少做事不做事的朴素的优化思想。同学们还记得骑单车到乘飞船的难忘之旅吧，仔细回想一下这是不是少做事不做事的经典案例。只是听明白体系结构原理而没有深入思考下去的人，他的单车永远无法变成飞船。”

2.3.3.2 老师的课程展望与规划

“下面我们进行一下未来课程的展望吧，未来大家如何学习，梁老师会继续给大家上什么？

收获，不止 Oracle

大家觉得要不要规划一下？”

“要！”

“首先请大家自行从 Oracle 官方下载 CONCEPT 文档，并尽量在一周的时间内阅读完毕（可先不阅读其中指向的细节部分，那是自己选择性阅读的情况）。此外可以阅读 Oracle 11g 也可以阅读 Oracle 10g 版本，这个请大家务必做到，我这里有 Oracle 10g 的官方文档 CONCEPT 的中英文对照文档，大家如果为了加快阅读速度，可以参考一下中文的说明，不过大家要记得英文阅读能力始终是工作中最重要的能力之一。

那我接下来会给大家上什么内容呢？请大家再次阅读角色分析图 2-29。

角色	基本原理					开发技能			管理知识					优化原理					设计相关					
	体系物理结构	体系逻辑结构	表	索引	事务	SQL	PL/SQL	常用函数	用户及权限管理	安装调试	备份恢复	数据迁移	闪回	故障处理	统计信息	执行计划	诊断工具	深入理解表	深入理解索引	表连接原理	模型工具使用	规范制定执行	业务理解	各类知识综合应用
开发	✓	✓	✓	✓	✓	✓	✓	✓								✓		✓	✓	✓				
管理	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓							
优化	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓				
设计	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

图 2-29 角色分析探讨

我的教学方法是有些特别的，绝不是为了讲述某个知识而讲述某知识，那样不仅效果不明显也无趣，我想教的东西是一个整体性的东西，是有启发性的东西，好比今天的课堂，大家应该已经感受到了。

基本原理这块还是接下来上课的重点，体系物理结构已经描述完毕了，剩下的就是体系逻辑结构和表及索引等的描述。接下来我们将描述开发技能与管理知识以及优化原理，同时对设计相关知识也会做一些描述。

哇，这么多课程该如何上啊，梁老师是不是太贪心了？”梁老师打趣地停了下来。

同学们会心地笑了一下，难忘的一天课程下来，他们早已被梁老师的用心和激情深深打动了，对梁老师很是尊重和钦佩。

“其实我不是来给大家上数据库的，主要是来谈理想，谈人生的。”

小莲不自主地点点了头表示赞许。

“关于课程的内容，五大领域都有涉及但是重心绝对不一样，基本原理将会占很大的篇幅，因为这是理解后面知识的基石。

我在这里郑重宣布：优化知识和优化思想将成为全部课程的中心内容，并结合各种案例融入各领域中去，如基本原理与优化、开发与优化、管理与优化、设计与优化。然后……然后还有优化的课程吗？”

“没了！”少数同学附和地应答了。

“是没了，但是从头到尾思想都贯穿其中，不只是具体的优化知识，更重要的是思想。我用心将后续的课程打造成一个充满世界观、方法论及精彩案例的心灵之旅，让大家能成为旅人而非学生。

基本原理中体系结构的学习大家都体验过了，后续老师将会延续这种风格和大家共度心灵之旅。虽然这种心灵之旅不是阅读官方文档所能达到的，但是梁老师的课程和官方文档却是需要互相补充、结合完善的。希望老师在精心准备课程的时间里，大家能抽空认真地选择性地阅读官方文档。今天的课程到此结束，期待我们的下次再见！”

台下响起了热烈而又长久的掌声。

第 3 章



神奇，走进逻辑体系世界

3.1 长幼有序的逻辑体系

数据库的逻辑体系如图 3-1 所示。

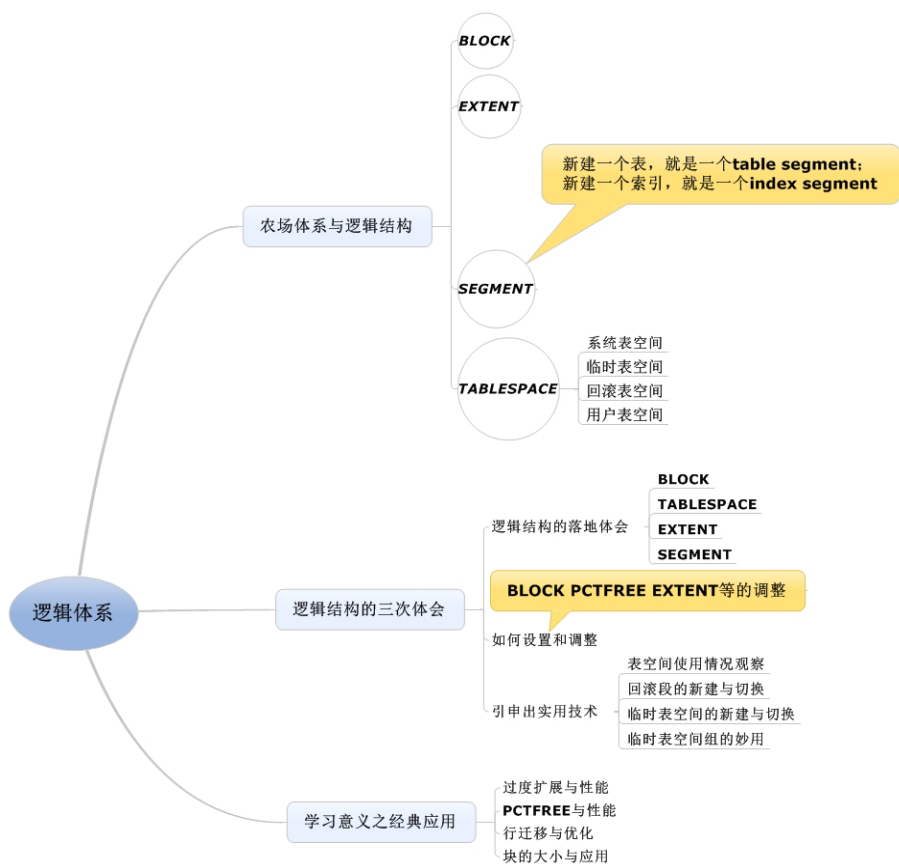


图 3-1 逻辑体系

3.2 逻辑体系从老余养殖细细说起

3.2.1 农场之体系逻辑结构

一周过去了，梁老师第二次上课的时间转眼就到了，大家都期盼已久，早早地坐在培训大厅里静静等候。

“我已经提早半小时了，你们居然都到齐了。”伴随着有些吃惊的声音，梁老师风风火火地走进培训大厅，第二课堂终于要拉开序幕了。

“上堂课我们讲述了学习路线图与体系物理结构，现在我们开始描述体系逻辑结构，什么是逻辑结构呢？”

大家先想想 Oracle 体系的物理结构是什么？那都是一些看得见摸到着的东西，上次的课程中我和大家一起登录数据库所在的主机，实实在在地体验了 SGA 共享内存段是如何被开辟而又如何消亡、后台进程是如何被唤起而又如何退出，此外也清楚地看到了数据文件、参数文件、控制文件、日志文件、归档文件的大小及位置。因此物理结构实质上可以理解为我们在物理上可以实实在在的看得见的东西。

而我们今天说的体系结构的逻辑结构正是从体系物理结构图中的数据文件部分展开描述的，如图 3-2 所示，请大家注意看圆圈标记处。

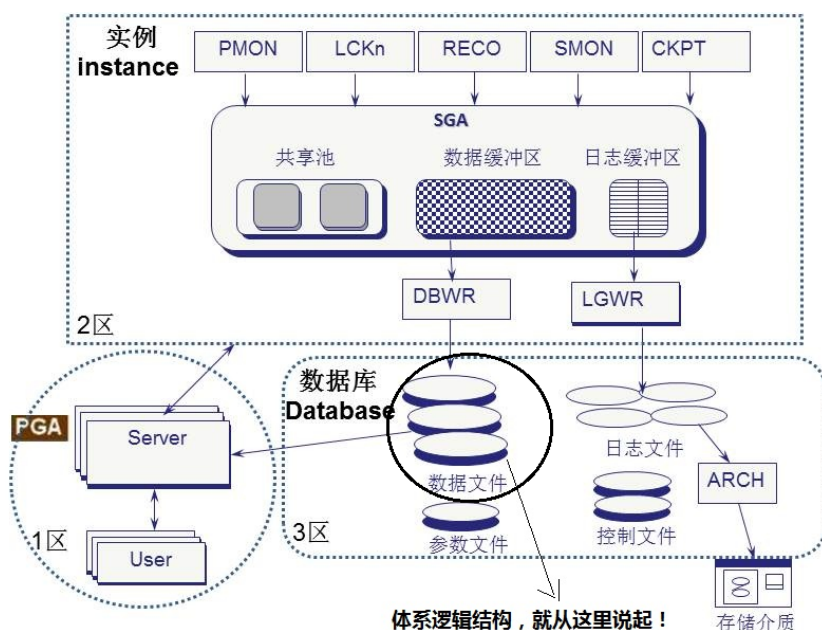


图 3-2 逻辑结构从此处说起

这里数据文件是存放数据之处，也是数据库存在的根本！不过这样的数据文件让我们使用者感觉有些无从下手了解，好像黑盒子一样。但是只要正确认识和剖析 Oracle 的体系逻辑结构，就可以让我们进一步加深对数据库的理解，不再是停留在黑盒子的认识水平，从而工作起来游刃有余。

这里我要隆重推出的逻辑结构是：表空间（TABLESPACE）、段（SEGMENT）、区（EXTENT）、块（BLOCK）。ORACLE SERVER 正是条理地通过表空间以及段、区、块控制磁盘空间的合理高效的使用，让我们观察图 3-3。

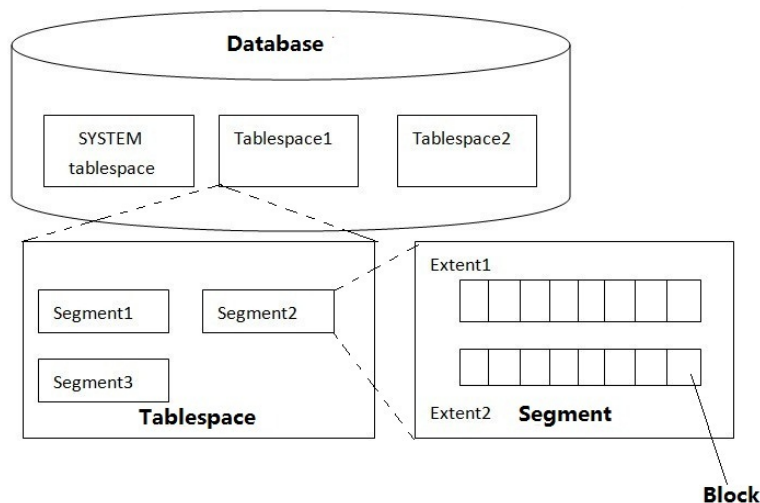


图 3-3 体系逻辑结构

从图 3-3 我们不难看出，数据库（DATABASE）由若干表空间（TABLESPACE）组成，表空间（TABLESPACE）由若干段（SEGMENT）组成，段（SEGMENT）由若干区（EXTENT）组成，区（EXTENT）又是由 Oracle 的最小单元块（BLOCK）组成的。

其中表空间又包含系统表空间、回滚段表空间、临时表空间、用户表空间。除了用户表空间外其他三种表空间有各自特定的用途，不可随意更改和破坏，尤其是系统表空间更是需要小心谨慎保护。

刚才我们是从大说到小，现在我们按照从小到大的方向再将它们描述一遍。

一系列连续的 BLOCK 组成了 EXTENT，一个或多个 EXTENT 组成了 SEGMENT，一个或多个 SEGMENT 组成了 TABLESPACE，而一个或多个 TABLESPACE 组成了 DATABASE（一个 DATABASE 想存在，至少需要有 SYSTEM 及 UNDO 表空间）。

说了这些，同学们明白吗？”梁老师停下来问同学们。

同学们都不出声，微微摇头听得有些似懂非懂。

“估计是让大家觉得太抽象了。”梁老师笑着说。

“大家还记得我在前面第一堂课上的内容吧，梁老师操作 DML 语句更新表，实质是更新 Oracle 磁盘的数据库文件（datafile），只是会有一个从 data buffer 中先更新再刷到磁盘的过程（直接路径方式除外），这个大家还记得吗？”

“记得！”同学们回答得很响亮。

“很好，能让大家记得这么牢，那是因为梁老师上体系物理结构时以具体的 `update t set object_id=92 where object_id=29` 为例子，对吧。这里其实就已经提及了 Oracle 体系结构中的逻辑结构，就是……”

“T 表！”晶晶很连贯地接下了梁老师的话。

“对！”梁老师肯定了晶晶的回答后继续往下说，“比如张三建了 T 表，李四建了 T2 表，王五建了 T3 表，这些动作肯定是写进数据文件 datafile 里，而你通过物理结构的认识就只是看见数据文件而已。

面对表操作肯定比面对数据文件直观形象得多，这个表是就从数据文件里直观抽象出来的逻辑结构。

前面我说了 Oracle 的逻辑结构从大到小分为表空间、段、区、数据库块这 4 部分。那张三、李四、王五建的表可以对应上述哪个逻辑结构？

其实是和段（SEGMENT）直接对应。其中 T 表就是 T 段，T2 表就是 T2 段，T3 表就是 T3 段（不过这里要注意，其实表并不是只对应一个段，如表中包含 LOB 类型列，则 LOB 至少会有两个段，数据段和索引段，如表有分区，则每个分区又都独立成段）。

如果建表的命令类似如下命令：

```
create table t (id int)    tablespace tbs_test;
create table t2 (id int)   tablespace tbs_test;
create table t3 (id int)   tablespace tbs_test;
```

那显而易见 T 段、T2 段、T3 段都属于 TBS_TEST 表空间，TBS_TEST 表空间是由 T 段、T2 段、T3 段等组成的。

到此大家对段有一个直观的认识了吧？”梁老师说到这里停了下来，看到同学们纷纷点头，梁老师继续说下去。

“前面描述的段（SEGMENT）是由区（EXTENT）组成的，而区又是由一系列数据块（BLOCK）组成的，我打个比方，假如一个区（EXTENT）固定是由 10 个数据块（BLOCK）组成的，我们插入 T 表的动作就是这样的：

数据插入某个区 EXTENT1 的第 1 个数据块中，很快插满了，然后就插入第 2 个同属于该 EXTENT1 区的连续数据块，接下来第 2 个块又插满了，插第 3 个块……当准备插第 11 个数据块时，这个区满了。接下来只好插入另外一个区 EXTENT2，直至插完 EXTENT3、EXTENT4 后，插入结束。

而这些 EXTENT1、EXTENT2、EXTENT3、EXTENT4 组成了 T 段，也就是张三的 T 表，至此，我把这 4 个逻辑结构说完了，大家能听明白吗，有什么疑问？”

“梁老师，为什么要有区（EXTENT）啊，不是说块（BLOCK）是数据库的最小单位吗，系列块直接组成 SEGMENT 不也一样吗，我觉得 EXTENT 不是必需的啊。”小莲一直在思考这个问题，立即举手提问了。

“问得好，同学们，小莲的这个疑问你们可以回答吗？”

台下鸦雀无声。

“看来没人回答啊，那好吧，我讲个故事来回答她的疑问行吗？”梁老师笑着说。

“好！”台下有些欢呼雀跃。

“故事还是和我们熟悉的老余一家有关。

在一个叫九峰山的山清水秀之地住着一群农夫，他们各自圈了一块很大的养殖场用于养殖牲畜，里面养着猪、牛、羊、兔等，不同种类牲畜又被围在各自的栅栏里分别圈养，不过这些场地的所有权都归王财主所有，农夫们只是租用。令人诧异的是，服装店生意经营红火的老余居然也成了这些农夫的一员。

原来老余的真正兴趣不在服装生意而是在农场养殖工作，因此他将生意交付给余太太全权管理，而他每天都进自己租用的养殖场做他感兴趣的养殖工作，分别到猪圈、牛圈、羊圈为牲畜们喂食。

由于这里水甜草美，所以这些牲畜的生长速度和繁殖速度非常快，其中最惊人的是这些猪，老余发现猪圈拥挤得几乎要被撑爆了，就急忙想扩大这个猪圈。

可是王财主和这些农夫有个奇怪的约定，就是虽然面积几万平米的养殖场都租给了农夫，可属于养殖场内的猪、牛、羊等牲畜圈要扩大还得让王财主知道，向他申请，而且一次只能申请增加 8 平米，还必须单独隔成一个小屋（现在老余的猪圈尽是由一个个 8 平米的小屋组成）。据说王财主这个奇怪的要求是因为他根据自己多年的经验了解到 8 平米小屋大致可容纳 10 只兔子、或者 4 头羊、或者 2 头猪、或者 1 头牛。由于每扩大 8 平米都要向他申请，他就可以通过记录牲畜圈里这些小屋的数目来预估农夫养殖的牲畜的大致数量，再从牲畜数量中预计其收成，从而作为后续租金调整的参考依据。因此在王财主心目中，8 平米这个单位是他的最小核算单位，非常重要！

老余找到了王财主，申请通过后在猪圈里扩建了一个 8 平米小屋，调整了几头猪进去，不过猪之家总体还是拥挤不堪，无奈老余又去找王财主，又扩建了一个 8 平米小屋，可惜还是拥挤，于是又去找王财主……

终于让猪们舒服点了，过几天，不少猪又大了，也有不少猪又生崽了，空间再次拥挤不堪，这下老余还得再去找王财主……

很快老余就累病倒了，他的儿子小余帮他想了一个好主意，就是每次找王财主申请不以 8 平

米为单位，而每次以 80 平米为单位申请，申请的次数显然会少很多，也就不会那么辛苦了。

于是老余找到王财主提出每次以 80 平米为单位进行申请。王财主开始不同意，后来转念一想，80 平米正好是 8 平米的 10 倍，可以认为每次申请意味着多了 100 只兔子、或者 40 头羊、或者 20 头猪、或者 10 头牛，好像并不影响自己的判断。而且最近因为频繁地会见这些不断来申请扩建的农夫，王财主累得疲惫不堪也快病倒了，想到这种改变必然会让自己轻松很多，于是王财主就答应了，不过他有个原则坚决不能改变，就是新增的 80 平米必须是新建出 10 个 8 平米的小屋。

遭遇同样烦恼的老黄、老林、老张等其他养殖户也因此受益了，他们开心地宴请了老余一家，以示感谢。

我的故事说完了，小莲，你还有疑问吗？”

“梁老师，我明白了，您说的这个猪圈从每次允许申请扩建 8 平米提升到每次可允许申请扩建 80 平米，这 80 平米的尺寸是暗指 EXTENT，而这 8 平米就暗指 BLOCK 吧。

Oracle 的这个 EXTENT 的设计是为了避免过度扩展，因为块的尺寸太小了，如果以这个块的尺寸为单位进行扩展，那就好比您在故事里说的每次以 8 平米为单位进行猪栏扩建一样，估计 Oracle 数据库也要像老余那样累病倒了。”

“回答得非常好，我来总结一下，王财主心目中的 8 平米是他的最小核算单元，类似 Oracle 的最小单位数据 (BLOCK)；小余建议的 80 平米类似 Oracle 的区 (EXTENT) 的概念；这些猪圈、牛圈、羊圈其实就是段 (SEGMENT)；这些牲畜组成的这个养殖场就是表空间 (TABLESPACE)；而老黄、老林、老张等其他养殖户的养殖场就是不同的 TABLESPACE，他们共同组成的九峰山养殖场家园就是数据库 (DATABASE)。

咦，老师发现，原来 Oracle 的这个 EXTENT 的设计是抄袭小余的思路啊。”

台下笑成一片，笑声中大家恍然大悟，至此 Oracle 这 4 个逻辑结构的相互关系连同猪圈的故事一起，被牢牢记在同学们脑海中了。

3.2.2 农场之 BLOCK 漫谈

“大家好，前面给大家大致描述逻辑结构组成时，大家明白了块、区、段、表空间这 4 部分的关系。现在我们将依次细说这些成员。先从 Oracle 逻辑结构的最小单位数据库块 BLOCK 说起。

虽然说 BLOCK 是 Oracle 的最小逻辑数据单位，但是所有数据在文件系统层面最小物理存储单位是字节，操作系统也有一个类似 Oracle 的块容量的参数 (block size)，但是 Oracle 总是访问整个 Oracle BLOCK，而不是按照操作系统的 block size 来访问的。

一般情况下大多数操作系统 OS 的块容量为 512 字节大小或其整数倍，而数据库块一般默认设置为 8KB，除此之外也有系统将其设置为 2KB、4KB、16KB、32KB、64KB 等其他大小。但是数据库的 BLOCK 一般要设置为操作系统 OS 块容量的整数倍，这样可以减少 IO 操作，这个大

家说说为什么？”梁老师忽然停了下来问大家。

台下沉默了一会儿，大家都在积极思考。

“我知道了，否则会浪费空间！”小莲起身说道，“数据库是运行在操作系统 OS 上的，而写入的数据文件也是在操作 OS 文件，因此真正操作的单位应该是操作系统 OS 块的大小。

假如 IO 的大小设置为 512 字节 (0.5KB)，本来如果 DB 的 BLOCK 设置为 1KB 正好是其 2 倍，但是设置为 0.8KB，这时由于操作系统的单个块大小为 0.5KB，只有 2 个操作系统块才可容纳下，于是就动用了 2 个 OS 块去容纳，相当于占用了 1KB 大小的 OS 空间，浪费了 0.2KB。”

“说得太好了。”梁老师赞许地点点头，继续往下说，“Oracle 的数据库块并不是简单地往里插数据，插满了装不下了就插入另一个数据块这么简单，而是额外提供了一定的管理功能。数据库的组成为数据块头（包括标准内容和可变内容）(common and variable header)、表目录区 (table directory)、行目录区 (row directory)、可用空间区 (free space)、行数据区 (row data) 这 5 个部分，具体如图 3-4 所示。

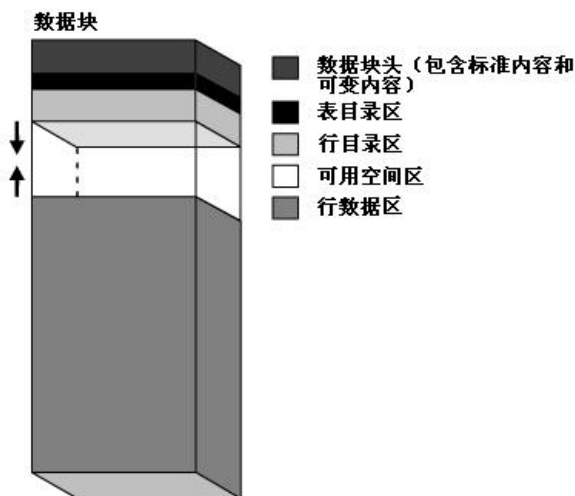


图 3-4 数据块结构

我把数据库块分成了 5 个部分，现在开始依次说明。

- ① 数据块头 (header) 中包含了此数据块的概要信息，例如块地址 (block address) 及此数据块所属的段 (segment) 的类型 (比如到底是表还是索引)。
- ② 表目录存放了什么？只要有一行数据插入到数据库块中，那该行数据所在的表的信息将被存储在这个区域。
- ③ 行目录存放什么？其实就是存放你插入的行的地址。
- ④ 可用空间区说来就简单了，就是块中的空余空间，为什么要有空余呢？后面我们会慢慢

道来。这个空余的多少由 Oracle 的 PCTFREE 参数设置，如果是 10，表示该块将会空余 10% 左右的空间。此外如果是表或者索引块，该区域还会存储事务条目，大致有 23 字节左右开销。

- ⑤ 而行数据区域更简单了，就是存储具体的行的信息或者索引的信息，这部分占用了数据块绝大部分的空间。

这里数据块头（data block header）、表目录区（table directory）、行目录区（row directory）被统称为管理开销（overhead），其中有些开销的容量是固定的，而有些开销的总容量是可变的。数据块中固定及可变管理开销的容量平均在 84 到 107 字节（byte）之间。

至此，我将 Oracle 的数据块结构说完了，大家都能理解吗？”

从台下同学们的表情来看，大家基本上都能明白。

“我接下来再说说农场的故事给大家加深一下印象吧，猪圈里的小房间类似于 Oracle 数据库块。比如你到了猪圈的某个小房间门前，你在门口上就看到门牌（标注了这个房间所在位置，如九峰山农场老余养殖场猪圈 6 号房），还可以看到标注了此房养 2 头猪（1 黑 1 白）的信息。同时还可以知道这两头猪的位置（一头在房间的左边，一头在右边），如果你推门进去，发现果然有 2 头猪。此外发现房间有一点空间预留着。

大家觉得是不是这个道理啊？”

“是！”同学们大声回答。

这段形象生动的描述让大家听得津津有味，印象也更深刻了。

3.2.3 农场之区与段

“大家都理解数据块了，接下来我们把一些连续的数据块（data block）组合在一起，就形成了区（EXTENT）。Oracle 中这个被称之为 EXTENT 的数据库逻辑存储分配单位就是这么形成的。

EXTENT 是 Oracle 数据库分配空间的最小单位，请注意分配这两个字眼。

请大家仔细体会，回想一下老余向王财主申请扩大猪圈的故事，一次申请 80 平米和一次申请 8 平米的区别，8 平米的小房间是农场的最小单位，而 80 平米是扩展空间的最小单位，你想扩大空间，必须以每次 80 平米为单位扩展，Oracle 的道理和农场的例子是一样的。说到这里，大家有什么疑问吗？”

“梁老师，Oracle 的区一般是由多少个连续的块组成的，每次申请的区一定是固定的大小吗，是系统默认值还是我能调整改变的？”小莲立即起身说出了自己的疑问。

“问得非常好，其他同学还有别的疑问吗？”梁老师继续问。

“梁老师，猪圈要养猪一开始就必须要有空间吧，应该是一开始有一个初始的大小，然后不够了再慢慢申请增加啊，Oracle 应该也是如此设计的吧？”晶晶有些不确信地问。

收获，不止 Oracle

“太好了，我就很喜欢同学们积极动脑思考提问，只有多思考印象才会深刻，下面我展开描述，大家认真听，相信提问的两位同学听完后就没有疑问了。

当某用户创建一张表 T 时，实质就是建了一个数据段 **segment T**（也就是猪圈开始建好了），如果是建 T2 表，可以理解为对应羊圈、兔圈等其他的，相信大家对这都有概念了。在 Oracle 数据库中，只要 **segment 创建成功**，数据库就一定为其分配了包含若干数据块（**data block**）的**初始数据扩展（initial extent）**，即便此时表中还没数据，但是这些初始数据扩展中的数据块已经为即将插入的数据做好准备，因此晶晶的猜想是正确的。

接下来 T 表（也就是 **SEGMENT T**）中开始插入数据，很快初始数据扩展中的数据块都装满了，而且又有新数据插入需要空间，此时 Oracle 会自动为这个段分配一个**新增数据扩展（incremental extent）**，这个新增数据扩展是一个段中已有数据扩展之后分配的后续数据扩展，容量大于或等于之前的数据扩展。

小莲同学很关心自己能否调整左右数据库的区的分配，其实是可以的。

每个段（**segment**）的定义中都包含了数据扩展（**extent**）的**存储参数（storage parameter）**。存储参数适用于各种类型的段。这个参数控制着 Oracle 如何为段分配可用空间。例如，用户可以在 **CREATE TABLE** 语句中使用 **STORAGE** 子句设定存储参数，决定创建表时为其数据段（**data segment**）分配多少初始空间，或限定一个表最多可以包含多少数据扩展。如果用户没有为表设定存储参数，那么表在创建时使用所在表空间（**tablespace**）的默认存储参数。

此外小莲还想了解分配的区的大小是否是固定的，其实这也可以由我们自主选择。

在一个本地管理的表空间中（注：还有一种数据字典管理的表空间，因为是一种要被淘汰的技术，这里就不提及了），其中所分配的数据扩展（**extent**）的容量既可以是用户设定的固定值，也可以是由系统自动决定的可变值。取决于用户创建 **tablespace** 时用 **UNIFORM** 指令（固定大小）还是 **AUTOALLOCATE** 指令（由系统管理）。

对于固定容量（**UNIFORM**）的数据扩展，用户可以为数据扩展设定容量（比如 100MB、1GB 等随你设定）或使用默认大小（1 MB）。用户必须确保每个数据扩展的容量至少能包含 5 个数据库块（**database block**）。本地管理（**locally managed**）的临时表空间（**temporary tablespace**）在分配数据扩展时只能使用此种方式。

对于由系统管理（**AUTOALLOCATE**）的数据扩展，你就无从插手干预了，Oracle 或许一个区申请 20M，下一个区忽然申请 100M，Oracle 在运行过程中自行决定新增数据扩展的最佳容量，你无从得知规律。不过还是有一个下限的，即区的扩展过程中其最小容量不能低于 64 KB，假如数据块容量大于等于 16 KB，这个下限将从 64KB 转变为 1 MB。

现在两位同学的疑问解决了没，大家都能理解吗？”

这节课结束，同学们进一步了解了区（**EXTENT**）的概念和规律。小莲闭上眼睛想回味一下

所学的知识，居然脑海中跳出农场老余焦虑地看着猪圈，着急想找王财主申请扩大猪圈空间的画面。她自己都觉得非常有趣，忍不住笑出声来了。

3.2.4 农场之表空间的分类

“同学们，区是由系列块组成的、段是由系列区组成的，那系列段又组成什么？”

“表空间！”同学们现在概念已经非常清晰了。

“大家知道表空间都有哪些类型吗？”

这下同学们都怔住了，有哪些类型？

“看来又要给大家说故事了。”梁老师笑起来。

“好！”台下又响起欢呼声。

“话说这个九峰山农场草肥水美，除了老余，还吸引了老黄、老林、老张等农夫在这里辛勤工作，虽然租金贵了点，但是让他们过上富足的日子还是不成问题。

王财主为人和善，大家都觉得比较好相处。不过他这个人比较讲原则，遇到他认为原则性的问题，是坚决不会让步的。比如具体到老余申请扩大猪圈那件事，他认为必须以8平米为最小单位在猪圈内建小屋子，就必须得这么办，没得商量。他认为非原则性的，比如一次申请80平米空间减少申请次数，他也就认可了，只要是保证那80平米能分成10间小屋子就成。

整个九峰山面积非常大，不过他却空余了不少地方不出租，无论农户开什么样的价钱都不为所动，遇到原则性问题，王财主可真是非常坚决。

其实王财主是用心良苦，绝不是故意有钱不赚。让我们来看看，他空余了许多地方不出租，都干啥用了。

3.2.4.1 表空间与系统农场

王财主在距老余农场、老黄农场等各农场都比较近的地方保留了一块空地，精心将其圈起来，并且起了一个响亮的名字：**系统农场**。这个农场可不养殖任何牲畜，里面存储了各农场需要搭建维修的必备工具，牲畜们的营养品、医疗保健用品，农夫的个人资料以及租用明细、王财主的收入清单，还有各个农场主农场的相关信息（养什么牲畜，养的数量，农场面积等）。

看来这个系统农场虽然没有养殖牲畜，却是非常重要的，对各个具体养殖的农场起到了管理和维护的作用，要是没有这个系统农场，各农场就无法正常经营下去，所以该系统农场存在的意义非同小可。

3.2.4.2 表空间与临时农场

王财主接下来还在老余、老黄等农场附近建了一个特别的农场，起名为**临时农场**。这个名字听起来有点不怎么正规，但是在王财主的眼中，却是非常重要的。这个农场平时不养牲畜的，但是

收获，不止 Oracle

却经常有牲畜来此活动，不过来了很快就离开了，只是一个临时活动的场所。看起来王财主很注重牲畜的健康，还提供了健身活动场所给它们，考虑得真周到啊。”

台下的同学爆发出一阵大笑，把梁老师的故事给打断了。

“王财主的临时农场能让牲畜们有多大的健身效果这不好说，老余却时常为了选出自己最大的几头猪而用到这个临时农场。比如某天早上，他把一群猪赶到宽阔的临时农场去排队，选出前排的 5 头大猪卖给上门收购的商人。

3.2.4.3 表空间与回滚农场

王财主还有第三个特殊的农场，命名为**回滚农场**，这名字听起来极特别，说起来却很简单。之前我们提到老余某天把群猪赶到临时农场，用排序的方法选出最大的 5 头猪想出售。当他选出这 5 头猪后，就会先将这 5 头猪驱赶到回滚农场待命。如果老余和上门的商人谈拢了价钱，5 头猪就让商人牵走了，要是谈不拢，老余就立即把猪赶回自己农场的猪圈，不卖了。

王财主的这三个特殊的农场虽然没有用来饲养牲畜，却为各个农夫的农场提供了方便，成了大家宝贵的公共资源。农夫们最终明白了这些道理后，都更加尊敬和感激王财主，大家相处得更加融洽了。

好了，到此故事说完了。同学们，表空间有什么类型我说好了，你们知道是哪些吗？”梁老师讲故事讲得口干舌燥，终于可以停下来喝口水了。

“**系统表空间、临时表空间、回滚表空间。**”敬昱起身回答。

“还要补充一个**数据表空间**吧，都没猪、羊可养了，王财主苦心经营的三个特殊农场为谁服务呢？”晶晶这小姑娘此话说得经典风趣，把大家都逗乐了。

“大家回答得非常好，梁老师非常满意，看来故事没白说。大家休息 10 分钟，回来我们继续探索体系逻辑结构的奥秘。”

3.2.5 逻辑结构之初次体会

“大家好，体系逻辑结构说完 4 小节内容了，大家是否觉得有些抽象？毕竟它们看不见摸不着。现在我通过对数据库的一系列操作，让大家实践体会一下。以下实验是通过观察数据库数据字典以及 `show parameter` 参数的方式来实现。

试验顺序是这样的：先建表空间（数据表空间、临时表空间和回滚段表空间），然后建指定用户，该用户登录后，在指定的表空间建表和建索引。

大家回去后，复习过程中可以用梁老师的实验脚本自行体会一下。

3.2.5.1 逻辑结构之 BLOCK

以下查询返回说明数据库的块（BLOCK）大小是 8KB，这是 Oracle 的最小逻辑单位。

```

sqlplus "/ as sysdba"
SQL> show parameter db_block_size
NAME                                TYPE                                VALUE
-----
db_block_size                       integer                             8192
----也可通过观察表空间视图 dba_tablespaces 的 block_size 值获取
select block_size
  from dba_tablespaces
 where tablespace_name='SYSTEM';
BLOCK_SIZE
-----
8192

```

脚本 3-1 查看 Oracle 块的大小

3.2.5.2 逻辑结构之 TABLESPACE

本来按顺序要先说块，再说区和段，但是考虑到先有表空间，后续才可以在这个表空间上做试验，所以这里我先说 TABLESPACE，请大家观察下面建各类表空间的方法：

--普通数据表空间

```

sqlplus "/ as sysdba"
SQL> create tablespace TBS_LJB
datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB_01.DBF' size 100M
extent management local
segment space management auto;
表空间已创建。
col file_name format a50
set linesize 366
SELECT file_name, tablespace_name, autoextensible,bytes
  FROM DBA_DATA_FILES
  WHERE TABLESPACE_NAME = 'TBS_LJB'
 order by substr(file_name, -12);

```

FILE_NAME	TABLESPACE_NAME	AUTOEXTENSIBLE	BYTES/1024/1024
E:\ORADATA\ORA10\DATAFILE\TBS_LJB_01.DBF	TBS_LJB	NO	100

---临时表空间（语法有些特别，有 TEMPORARY 及 TEMPFILE 的关键字）

```

CREATE TEMPORARY TABLESPACE temp_ljb
  TEMPFILE 'E:\ORADATA\ORA10\DATAFILE\TMP_LJB.DBF' SIZE 100M;
表空间已创建。
SQL> SELECT FILE_NAME,BYTES,AUTOEXTENSIBLE FROM DBA_TEMP_FILES where tablespace_name='TEMP_LJB';

```

FILE_NAME	BYTES	AUTOEXTENSIBLE
E:\ORADATA\ORA10\DATAFILE\TMP_LJB.DBF	104857600	NO

---回滚段表空间（语法有些特别，有 UNDO 的关键字）

SQL> create **undo** tablespace undotbs2 **datafile** 'E:\ORADATA\ORA10\DATAFILE\UNDOTBS2.DBF' size 100M;
表空间已创建。

```
SELECT file_name,
       tablespace_name,
       autoextensible,
       bytes/1024/1024
       FROM DBA_DATA_FILES
       WHERE TABLESPACE_NAME = 'UNDOTBS2'
       order by substr(file_name, -12);
```

FILE_NAME	TABLESPACE_NAME	AUTOEXTENSIBLE	BYTES/1024/1024
E:\ORADATA\ORA10\DATAFILE\UNDOTBS2.DBF	UNDOTBS2	NO	100

---系统表空间（Oracle 10g 的系统表空间还增加了 SYSAUX 作为辅助系统表空间使用）

```
SELECT file_name,
       tablespace_name,
       autoextensible,bytes/1024/1024
       FROM DBA_DATA_FILES
       WHERE TABLESPACE_NAME LIKE 'SYS%'
       order by substr(file_name, -12);
```

FILE_NAME	TABLESPACE_NAME	AUTOEXTENSIBLE	BYTES
E:\ORADATA\ORA10\DATAFILE\O1_MF_SYSTEM_7FYXFMOM_.DBF	SYSTEM	YES	580
E:\ORADATA\ORA10\DATAFILE\O1_MF_SYSAUX_7FYXG03J_.DBF	SYSAUX	YES	360

---系统表空间 and 用户表空间都属于永久保留内容的表空间

```
select tablespace_name,
       contents
       from dba_tablespaces
       where tablespace_name in
       ('TBS_LJB', 'TEMP_LJB', 'UNDOTBS2', 'SYSTEM', 'SYSAUX');
```

TABLESPACE_NAME	CONTENTS
SYSAUX	PERMANENT
SYSTEM	PERMANENT
TBS_LJB	PERMANENT
TEMP_LJB	TEMPORARY
UNDOTBS2	UNDO

脚本 3-2 查看 Oracle 数据、临时、回滚、系统表空间情况

3.2.5.3 逻辑结构之 USER

如果你希望指定用户的默认表空间是某指定表空间，那该表空间必须先建立，现在我们参与测试的表空间已经建好了，用户就可以指定默认表空间在该表空间了，请大家记得试验的先后顺序。

```

---sysdba 用户登录，假如 ljb 用户存在，先删除
sqlplus "/ as sysdba"
drop user ljb cascade;
---建用户，并将先前建的表空间 tbs_ljb 和临时表空间 temp_ljb 作为 ljb 用户的默认使用空间。
create user ljb
identified by ljb
default tablespace tbs_ljb
temporary tablespace temp_ljb;
---授权，暂且将最大权限给 ljb 用户（大家切记只能在非生产环境做实验）
grant dba to ljb;
---可以登录 ljb 用户了
connect ljb/ljb

```

脚本 3-3 Oracle 建用户和授权简单体验

3.2.5.4 逻辑结构之 EXTENT

Oracle 的最小逻辑单位是块（BLOCK），而最小的扩展单位是区（EXTENT），这两个‘最小’请务必牢记。

```

---构造 t（注，如果没有指明表空间，就是用户 ljb 的默认表空间）
sqlplus ljb/ljb
drop table t purge;
create table t (id int) tablespace tbs_ljb;
---查询数据字典获取 extent 相关信息
select segment_name,
extent_id,
tablespace_name,
bytes/1024/1024,blocks
from user_extents
where segment_name='T';

```

SEGMENT_NAME	EXTENT_ID	TABLESPACE_NAME	BYTES/1024/1024	BLOCKS
T	0	TBS_LJB	0.0625	8

```

---插入数据后继续观察,发现由原来的 1 个区增加为 28 个区
insert into t select rownum from dual connect by level<=1000000;
commit;
select segment_name,

```

收获，不止 Oracle

```
extent_id,  
bytes/1024/1024,blocks  
from user_extents  
where segment_name='T' ;  
SEGMENT_NAME          EXTENT_ID    BYTES/1024/1024    BLOCKS  
-----  
T                      0           0.0625             8  
T                      1           0.0625             8  
T                      2           0.0625             8  
T                      3           0.0625             8  
T                      4           0.0625             8  
T                      5           0.0625             8  
T                      6           0.0625             8  
T                      7           0.0625             8  
T                      8           0.0625             8  
T                      9           0.0625             8  
T                     10           0.0625             8  
T                     11           0.0625             8  
T                     12           0.0625             8  
T                     13           0.0625             8  
T                     14           0.0625             8  
T                     15           0.0625             8  
T                     16            1            128  
T                     17            1            128  
T                     18            1            128  
T                     19            1            128  
T                     20            1            128  
T                     21            1            128  
T                     22            1            128  
T                     23            1            128  
T                     24            1            128  
T                     25            1            128  
T                     26            1            128  
T                     27            1            128  
已选择 28 行。
```

脚本 3-4 Oracle 的 extent 体会

3.2.5.5 逻辑结构之 SEGMENT

---构造 t 表

```
sqlplus ljb/ljb  
drop table t purge;
```

```
create table t (id int) tablespace tbs_ljb;
---查询数据字典获取 segment 相关信息
```

```
select segment_name,
segment_type,
tablespace_name,
blocks,
extents,bytes/1024/1024
from user_segments
```

```
where segment_name = 'T';
```

SEGMENT_NAME	SEGMENT_TYPE	TABLESPACE_NAME	BLOCKS	EXTENTS	BYTES/1024/1024
T	TABLE	TBS_LJB	8	1	0.0625

```
---插入数据后继续观察
```

```
insert into t select rownum from dual connect by level<=1000000;
commit;
```

---插入大量记录后，发现确实有变化，BLOCKS 和 EXTENTS 都增加了，区从 1 个增加为 28 个，块个数由原来的 8 个增加为 1664 个，段的大小从 0.0625MB 增长为 13MB，具体如下：

```
select segment_name, segment_type, tablespace_name, blocks, extents, bytes/1024/1024
from user_segments where segment_name = 'T';
```

SEGMENT_NAME	SEGMENT_TYPE	TABLESPACE_NAME	BLOCKS	EXTENTS	BYTES/1024/1024
T	TABLE	TBS_LJB	1664	28	13

```
---观察索引段（其中 IDX_ID 这个段的 segment_type 为 INDEX）
```

```
SQL> create index idx_id on t(id);
```

索引已创建。

```
select segment_name,
segment_type,
tablespace_name,
blocks,
extents,
bytes/1024/1024
from user_segments
```

```
where segment_name = 'IDX_ID';
```

SEGMENT_NAME	SEGMENT_TYPE	TABLESPACE_NAME	BLOCKS	EXTENTS	BYTES/1024/1024
IDX_ID	INDEX	TBS_LJB	2048	31	16

```
SQL> select count(*) from user_extents WHERE segment_name='IDX_ID';
COUNT(*)
```

```
-----
31
```

脚本 3-5 Oracle 的 segment 体会

通过以上试验，大家应该可以更直观地体会到 Oracle 的逻辑结构，这些试验大家都能看明白吗？”

3.2.6 逻辑结构之二次体会

3.2.6.1 BLOCK 的大小与调整

“梁老师，您之前的试验中数据库块 BLOCK 的大小我们可以调整吗？”林君举手问。

“问得好！”梁老师笑着说，“一般来说，Oracle 默认的数据库块大小就是 8KB，是你在创建数据库时决定的，所以如果想改变块的大小，就必须在建库时指定。

Oracle 9i 以后的版本中，Oracle 支持用户在新建用户表空间时指定块的大小，这意味着你的数据库有多个表空间，他们各自的 BLOCK 大小有可能各不相同。

切记只是新建自己的用户表空间，我们不可能更改原有的已经建好的表空间，系统表空间更不可能更改或调整，我们来看一个 Oracle 的参数设置，如下：

```
SQL> show parameter cache_size
```

NAME	TYPE	VALUE
db_16k_cache_size	big integer	0
db_2k_cache_size	big integer	0
db_32k_cache_size	big integer	0
db_4k_cache_size	big integer	0
db_8k_cache_size	big integer	0
db_cache_size	big integer	0
db_keep_cache_size	big integer	0
db_recycle_cache_size	big integer	0

脚本 3-6 Oracle 可启用不同大小的块

这里意味着你可以设置 2KB、4KB、8KB、16KB、32KB 的块大小，当你把类似 db_16k_cache_size 设置为 100MB，就意味着 SGA 中的 DATA BUFFER 数据缓存区中将会有 100MB 的大小让内存块可以以 16KB 的大小进行访问了，同时也意味着 16KB 大小的设置从此生效了。

现在我们做个测试，设置一个表空间，让其 BLOCK_SIZE 尺寸为 16KB，首先需要将 db_16k_cache_size 取值设置为非空：

```
SQL> alter system set db_16k_cache_size=100M;
```

系统已更改。

```
SQL> show parameter 16k
```

NAME	TYPE	VALUE

```
db_16k_cache_size          big integer    100M
```

脚本 3-7 启用 BLOCK_SIZE 为 16K 的块

然后建表空间，切记加上 **blocksize 16K** 的关键字即可，如下：

```
SQL>create tablespace TBS_LJB_16k
blocksize 16K
datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB_16k_01.DBF' size 100M
autoextend on
extent management local
segment space management auto;
表空间已创建。

---观察发现，TBS_LJB_16K 这个表空间果然不同于原来的 TBS_LJB2 表空间的，块的大小果然为 16K
select tablespace_name,
block_size
  from dba_tablespaces
 where tablespace_name in ('TBS_LJB2','TBS_LJB_16K');
TABLESPACE_NAME          BLOCK_SIZE
-----
TBS_LJB2                  8192
TBS_LJB_16K              16384
```

脚本 3-8 启动大小为 16K 的块新建表空间

好了，林君，听明白了吗？”

“明白了，谢谢梁老师！”

3.2.6.2 PCTFREE 参数与调整

“梁老师，我有疑问，您之前说 BLOCK 块有一个 FREE 空间，是由 PCTFREE 参数决定的，设置这个参数来控制 BLOCK 保留一些空间有啥意义啊，装得满满的不是更好吗？”小莲问。

“很好，我想问大家，数据库中有某表 T 有 900 行记录，如果一个块最多可以装 10 行记录，最终需要 90 个块将 T 表记录装满。如果 PCTFREE 为 10，表示会预留 10% 的空间，那就是每个块都只能装 9 行数据，最终需要 100 个块才可以把 T 表记录装满。

这时我们做一个全表扫描的查询，查询 T 表的所有记录，如果 PCTFREE 设置为 10，我们将会遍历 100 个数据块，如果为 0，我们将遍历 90 个数据块，大家觉得哪种设置性能更高？”

“当然是 PCTFREE 设置为 0 了。”小莲脱口而出，其实她之前就是这么想的，所以才提问的，梁老师这么一说，她更觉得就是如此了。

“说得好，如果 PCTFREE 设置为 0，块将要装下更多行的记录，需要的块的数量就少了，但是只有在只读数据库或者说只有插入删除很少更新的数据库环境中，才合适将 PCTFREE 设置为

0，那什么时候需要空余空间呢？下面梁老师讲述一个故事，大家就明白了，听吗？”

“听！”台下传来一片欢呼声。

“有一栋学生宿舍住着不少学生，501 房间住了小余在内的 4 个学生，他们生活得还算安宁和谐，可是小余不注意锻炼又贪吃贪睡，慢慢变成了一个大胖子，很快 501 拥挤得无法生活了。这时小余只好搬离 501，来到了有空余能容纳他的 502。小余没有自我反省，反而越吃越多，最后连 502 都住不下了，而 501 依旧没有空间可以容纳自己，最好他只好搬进了还有空余空间可以容纳他的 503，好了，故事说完了。”

“不是吧，这是什么故事啊！”台下有人笑着抗议了。

“看来不好听啊。”梁老师笑着说，“好吧，那我继续吧。小余的朋友小陈来学校找小余，小陈通过宿舍楼传达室记录的学生地址，直接来到了 501 房间找小余，结果发现门牌上写着小余已搬家到 502。于是他就去了 502 房间，发现上面还有挂着一个牌子，上面写着小余已经搬家到 503，最后在 503 房间终于找到小余了，不过这次花费了比往常更多的时间。

小陈开始埋怨小余，找你真辛苦啊。小余也很无奈，他也埋怨宿舍的剩余空间太少了，而他又改变不了自己贪吃的毛病，每次变胖就要因为太挤被人赶出去。

小陈也对小余深表同情，不过他还是安慰了一下小余，说学校还算好了，要是你们一开始 4 个人就是住得满满的，一点空间都不留，你岂不是更要频繁搬家了，好歹学校还预留了一点空间给你了，不是吗？

小余还是有些难过，不过过了些天，小陈接到小余的电话，电话那头小余很开心，他说他虽然又更胖了，不过他居然回到 501 了，看到了老同学他真高兴，而且这下很多人可以直接根据地址就找到他了。

原来小余在 503 又住不下，宿舍的制度是让他回到最早的房间去查是否有位置，假如 501 住不下他就找 502，502 如果也没有，他就去找其他房间了，结果他这次很幸运，回到 501 宿舍了。

故事说完了，块的属性 PCTFREE 设置也就说完了。小莲同学，你应该也完全理解这个参数该如何设置了吧？”

3.2.6.3 PCTFREE 与生效范围

“梁老师，您说得太形象了，我明白了，不过我还想问一个问题，这个参数是对整个数据库生效，还是只针对某个具体段的系列区包含的 BLOCK 生效？”小莲问。

“那我再说故事吧。小余的宿舍楼是男生宿舍，而这宿舍的背后是女生宿舍，这些女生特别爱美，注意饮食也注意锻炼，身材都保持得非常好，女生们从入学到毕业体重几乎都没啥变化。考虑到这个因素，再考虑到宿舍楼很紧张，所以学校把女生宿舍房间安排得满满的，几乎没留什么空余。不过女生宿舍因为拥挤住不下而搬家的情况却从没遇见过。我说完了，你明白我的意思了吗？”

小莲还是有些不明白。她觉得梁老师说的，她不是已经明白了吗？

“梁老师，我知道了，块的设置属性可以视每个段的具体情况进行不同设置，因为女生宿舍和男生宿舍的情况明显不同，如果设置一样的空余空间就不太合理，女生宿舍可以留少量空余空间甚至不留，而贪吃贪睡的男生宿舍却要多留。

好比我们数据库中的表一样，有的表频繁更新，有的表几乎是只读的，从不更新。我觉得不同类型的表就应该设置不同的 PCTFREE，表在数据库中就是 SEGMENT，因此这个参数其实是可以只针对某个具体段的系列区包含的 BLOCK 生效。”晶晶见到小莲半天不回答，忍不住抢答了。

“说得非常好！应该说 Oracle 有一个默认的属性，就是 PCTFREE=10，我们可以理解这个参数是在数据库层面生效的，但是具体到建 T 表时如果你指定 PCTFREE 为别的值，比如 20，那你这个 T 表或者说 SEGMENT T 的所有块的属性，就是 PCTFREE 为 20。大家觉得 Oracle 这样的机制合理吗？”

“合理！”小莲脱口而出，她看到大家都还没回答，显得有些不好意思了。

3.2.6.4 EXTENT 尺寸与调整

“大家应该都知道梁老师上课的特点了，我上课一般先说最主要的、必须首先掌握的知识，而扩展及延伸、细节和应用一般都不会立即说，这些话题我视为给大家思考的宝贵机会。大家仔细思考老师前面做的试验，有什么疑问吗？”

“对了梁老师，之前您有说过区的大小是可以设置的，不过我看到您在逻辑结构与 EXTENT 体会的描述中，有时是 0.0625MB（扩展 8 个块），有时是 1MB（128 个块），觉得我们并不能控制啊。”敬昱问。

“问得好，当初区大小的设置是由小莲同学提出来的，我还以为小莲会问这个问题。”梁老师笑着说。

小莲也笑了。

“其实之前建 TBS_LJB 表空间时 EXTENT 的设置是自动扩展的，如果我们要干预也可以，现在我再建一个表空间 TBS_LJB2，大家注意我的参数设置中 **uniform size 10M** 的关键字，这表示扩展是统一尺寸，大小都是 10MB。我们也可不指定大小，默认为 1MB，具体如下：

```
SQL> create tablespace TBS_LJB2
datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB2_01.DBF' size 100M
extent management local
uniform size 10M
segment space management auto;
表空间已创建。
```

脚本 3-9 建 UNIFORM SIZE 为 10M 的表空间

接下来建一个指定在 TBS_LJB 表空间上的表 T2，然后观察区的分配情况，如下：


```
sqlplus ljb/ljb
SQL> create table t2 (id int ) tablespace TBS_LJB2;
表已创建。
---观察 EXTENT 的分配情况
select segment_name,
extent_id,
tablespace_name,
bytes/1024/1024,blocks
from user_extents
where segment_name='T2';
```

SEGMENT_NAME	EXTENT_ID	TABLESPACE_NAME	BYTES/1024/1024	BLOCKS
T2	0	TBS_LJB	10M	1280

```
--接下来继续插入数据
SQL> insert into t2 select rownum from dual connect by level<=1000000;
已创建 1000000 行。
SQL> commit;
提交完成。
--再观察 EXTENT 的分配情况
select segment_name,
extent_id,
tablespace_name,
bytes/1024/1024,blocks
from user_extents
where segment_name='T2';
```

SEGMENT_NAME	EXTENT_ID	TABLESPACE_NAME	BYTES/1024/1024	BLOCKS
T2	0	TBS_LJB	10M	1280
T2	1	TBS_LJB	10M	1280

脚本 3-10 观察 UNIFORM SIZE 为 10MB 的表空间的分配情况

敬昱同学，听明白了吗？”

敬昱满意地点点了头。

“老师试验的结果很清晰，大家明白应不困难，难的是如何学会让自己积极去思考。”

3.2.7 逻辑结构之三次体会

3.2.7.1 已用与未用表空间情况

“看到大家积极思考，踊跃提问，梁老师非常满意！还有同学有其他疑问吗？”

“梁老师，您建的 TBS_LJB 表空间有 100MB，后来您又对 TBS_LJB 表空间下的 T 表插入 100 万条记录，我想知道现 TBS_LJB 表空间已使用多少，还剩多少，这如何得知呢？”小莲问。

“问得非常好，那我先回答这个问题吧，其实观察剩余多少，我们可以通过数据字典 dba_free_space 获取到的，这个数据字典可以得知表空间剩余多少的记录，具体如下：

```
select sum(bytes) / 1024 / 1024
      from dba_free_space
     where tablespace_name = 'TBS_LJB';
SUM(BYTES)/1024/1024
-----
        68.9
```

脚本 3-11 观察表空间的剩余情况

这里说明了 TBS_LJB 使用了 68.9MB，那剩余多少呢？”

“21.1MB”小莲对数学最敏感，脱口而出。

“小莲是知道老师之前建的表空间是 100MB 吧，要是老师没告诉你呢？”

小莲不好意思地笑了笑。

“使用了多少其实挺简单的，知道原始表空间是多大，相减就是使用了多少，原始表空间多大，其实在之前的逻辑结构与初次体会中，已经描述过了，通过 DBA_DATA_FILES 即可获得。

```
select  sum(bytes) / 1024 / 1024
      from dba_data_files
     where tablespace_name = 'TBS_LJB';
BYTES/1024/1024
-----
        100
```

脚本 3-12 观察表空间的总体分配情况

这下小莲的问题解决了吧， $100-68.9=21.1$ MB 就是已使用的表空间大小了，小莲同学，这个疑问解决了吧？”

“谢谢梁老师！”

3.2.7.2 表空间大小与自动扩展

“梁老师，我有疑问！”晶晶举手问，“我怎么看到您建 TBS_LJB 表空间后从 DBA_DATA_FILES 里观察到 AUTOEXTENSIBLE 字段的取值为 NO，是否数据超过 100MB，TBS_LJB 就无法扩展而装不下数据，那怎么办呢？”

“晶晶同学观察得很仔细，DBA_DATA_FILES 里观察到 AUTOEXTENSIBLE 字段的取值为 NO 确实表示表空间无法自动扩展。那意味着如果我们继续插入数据，超过 100MB 将会出错。下

收获，不止 Oracle

面我试验一下多次插入 T 表记录，最终报错的场景：

```
SQL> insert into t select rownum from dual connect by level<=1000000;
1000000 rows inserted
SQL> commit;
Commit complete
SQL> insert into t select rownum from dual connect by level<=1000000;
ORA-01654: 索引 LJB.IDX_ID 无法通过 1024 (在表空间 TBS_LJB 中) 扩展
```

脚本 3-13 不断插入记录，模拟表空间不足的场景

这里是我们观察一下 TBS_LJB 空间剩余多少了，只剩下 1.9MB 了，当然装不下 T 表数据及 T 表的索引了：

```
select sum(bytes) / 1024 / 1024
      from dba_free_space
     where tablespace_name = 'TBS_LJB';
SUM(BYTES)/1024/1024
-----
1.9375
```

脚本 3-14 表空间不足报错时再观察一下表空间剩余情况

那现在怎么办？方法其实有两种，一种是增加数据文件，扩大表空间。另一种就是把表空间设置为自动扩展。我们先试验一下增加数据文件的方式。

以下命令是扩大数据库表空间的方式。

```
SQL> ALTER TABLESPACE TBS_LJB
      ADD DATAFILE 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB_02.DBF' SIZE 100M;
Tablespace altered
```

脚本 3-15 表空间扩大的方法

表空间剩余空间变为 101MB 了，这下够大了。

```
select sum(bytes) / 1024 / 1024
      from dba_free_space
     where tablespace_name = 'TBS_LJB';
SUM(BYTES)/1024/1024
-----
101.875
```

脚本 3-16 表空间扩大后继续观察剩余空间情况

不过很显然的是，如果再多次插入数据，TBS_LJB 表空间很快就又会装不下了，因为虽然空

间是 100+100=200MB 了，但是 AUTOEXTENSIBLE 的属性都是 NO，都是不能自动扩展的。

```
SQL> col file_name format a50
```

```
SQL> SELECT file_name,
           tablespace_name,
           autoextensible,bytes/1024/1024
           FROM DBA_DATA_FILES
           WHERE TABLESPACE_NAME = 'TBS_LJB';
```

FILE_NAME	TABLESPACE_NAME	AUTOEXTENSIBLE	BYTES/1024/1024
E:\ORADATA\ORA10\DATAFILE\TBS_LJB_01.DBF	TBS_LJB	NO	100
E:\ORADATA\ORA10\DATAFILE\TBS_LJB_02.DBF	TBS_LJB	NO	100

脚本 3-17 观察表空间是否是自动扩展的

这时我们不必再试验插入数据了，肯定是一开始可以插入成功，再来几下就又装不下了，我们试验一下修改为表空间自动扩展方式。

```
SQL> alter database datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB_02.DBF' autoextend on;
Database altered
```

脚本 3-18 将表空间属性更改为自动扩展

现在再观察，TBS_LJB_02.DBF 属性 AUTOEXTENSIBLE 已经变化为 YES 了。

```
col file_name format a50
```

```
SELECT file_name,
       tablespace_name,
       autoextensible,
       bytes/1024/1024
       FROM DBA_DATA_FILES
       WHERE TABLESPACE_NAME = 'TBS_LJB';
```

FILE_NAME	TABLESPACE_NAME	AUTOEXTENSIBLE	BYTES/1024/1024
E:\ORADATA\ORA10\DATAFILE\TBS_LJB_01.DBF	TBS_LJB	NO	100
E:\ORADATA\ORA10\DATAFILE\TBS_LJB_02.DBF	TBS_LJB	YES	100

脚本 3-19 继续查看表空间属性，发现已经更改为自动扩展

这下即使我们反复插入数据，也不用担心空间不足了，果然我们观察到 E:\ORADATA\ORA10\DATAFILE\TBS_LJB_02.DBF 从原来的 100MB 自动扩大到了 132MB。

```
SQL> insert into t select rownum from dual connect by level<=1000000;
1000000 rows inserted
SQL> insert into t select rownum from dual connect by level<=1000000;
```

收获，不止 Oracle

```
1000000 rows inserted
SQL> insert into t select rownum from dual connect by level<=1000000;
1000000 rows inserted
SQL> insert into t select rownum from dual connect by level<=1000000;
1000000 rows inserted
SQL> commit;
Commit complete

SELECT file_name,
tablespace_name,
autoextensible,
bytes/1024/1024
  FROM DBA_DATA_FILES
 WHERE TABLESPACE_NAME = 'TBS_LJB';
```

FILE_NAME	TABLESPACE_NAME	AUTOEXTENSIBLE	BYTES/1024/1024
E:\ORADATA\ORA10\DATAFILE\TBS_LJB_01.DBF	TBS_LJB	NO	100
E:\ORADATA\ORA10\DATAFILE\TBS_LJB_02.DBF	TBS_LJB	YES	132

脚本 3-20 自动扩展后不用担心表空间不足，不过也要小心磁盘空间情况

好了，说到这里，晶晶同学，听明白了吗？”

“那在初次建表空间时，是否可以立即就设置为自动扩展呢，不要后续去修改，这样行吗？”
晶晶继续提问。

“可以的，和 `alter` 数据文件属性的关键字一样，加上 **autoextend on** 的关键字即可，老师做一个试验，顺道教大家删除表空间的语法，其中 `including contents and datafiles` 表示要删除表空间的数据和对应的数据文件，如果表空间有数据，不增加 `including contents` 将无法删除成功（Linux 及 UNIX 操作系统下增加 **and datafiles** 关键字可自动删除数据文件，而 Windows 环境下需要手工删除），具体脚本如下：

```
drop tablespace TBS_LJB
including contents and datafiles;

create tablespace TBS_LJB
datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB_01.DBF' size 100M
autoextend on
extent management local
segment space management auto;
```

脚本 3-21 删除表空间自动删除数据文件方法

以上脚本老师就不执行了，晶晶同学，你看明白了吗？”

“谢谢梁老师!”

“你们这么善于思考，我要谢谢你们!” 梁老师笑着称赞大家，“此外大家注意如下的写法，这里的 next 64k 表示每次都以 64KB 进行扩展，基本上等同于 uniform 64k 的功能。此外大家要记住可以控制最大表空间尺寸，比如如下命令即表示最大不能超过 5GB:

```
create tablespace TBS_LJB3
datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB3_01.DBF' size 100M
autoextend on
next 64k
maxsize 5G;
```

脚本 3-22 建自动扩展表空间可控制最大扩展到多少

好了，还有谁有疑问呢，请积极思考，继续提问。”

“梁老师，为什么要您在前面建表空间时需要增加 **extent management local** 和 **segment space management auto**;这些关键字啊，而您最新这次建表空间却没有加，这是为什么，这些关键字是什么含义？”小莲有些不解地问。

“很好，老师就知道有人会问这个问题，终于等到了。”梁老师笑着说，“其实如果你的数据库版本是 Oracle 10g 及以上版本，这两行命令是可以取消的，系统默认都是区的本地管理和段的自动管理，早期版本中，区的管理是依据数据字典的，导致系统产生大量的递归调用，随后的版本改为通过区上的位图标记来管理区的扩展，性能得以极大提升。Oracle 9i 时 Oracle 提供了这个新功能，而 10g 以上完全取消字典管理功能。由于时间有限，又有太多重要的其他知识需要描述，这里的技术演变的细节就不做描述了。

不过由于现在 Oracle 数据库已经不提供对 Oracle 9i 及以下版本的支持了，所以大家在编写建立表空间的代码中，可以将这两行去掉，我写这个的目的有两点：一来看大家的观察能力；二来是让大家知道在操作数据库时，要留意数据库版本，这也是分析问题的一个很重要的环节。”

3.2.7.3 回滚表空间新建与切换

“梁老师，我看到您建回滚表空间和临时表空间的操作，觉得很奇怪，怎么还需要我们自己建呢？我从这两个表空间提供的功能来看，猜测应该是系统自带的表空间啊。”曾祥非常疑惑地问。

“非常好，学习如何才能进步，源于细心观察加上认真思考。”梁老师欣慰地笑了，“曾祥的猜测是对的，Oracle 数据库建好后，UNDO 表空间和 TEMP 表空间必然是建好了。但是实际情况是，回滚段和表空间都可以新建，并且用户都可以指定新建的空间，我们先从回滚表空间开始分析。

通过以下命令得知，数据库当前的回滚表空间名为 UNDOTBS1:

收获，不止 Oracle

```
sqlplus "/ as sysdba"
```

```
SQL> show parameter undo
```

NAME	TYPE	VALUE
undo_management	string	AUTO
undo_retention	integer	900
undo_tablespace	string	UNDOTBS1

--其中 undo_management 的取值为 AUTO 表示是系统自动管理表空间而非手动管理。

脚本 3-23 查看数据库当前在用回滚段

查看当前数据库建有两个回滚表空间，并且状态 STATUS 都是有效的，其中 UNDOTBS2 就是在“逻辑结构之 TABLESPACE”这一小节描述时建成的。

```
SQL> select tablespace_name,status from dba_tablespaces where contents='UNDO';
```

TABLESPACE_NAME	STATUS
UNDOTBS1	ONLINE
UNDOTBS2	ONLINE

脚本 3-24 查看数据库有几个回滚段

分别查看两个回滚表空间的大小，发现系统自带的 UNDOTBS1 表空间已经有 5GB 这么大了：

```
SQL> select tablespace_name,
       sum(bytes) / 1024 / 1024
       from dba_data_files
       where tablespace_name in ('UNDOTBS1', 'UNDOTBS2')
       group by tablespace_name;
```

TABLESPACE_NAME	SUM(BYTES)/1024/1024
UNDOTBS1	5515
UNDOTBS2	100

脚本 3-25 查看数据库有几个回滚段，并得出它们的大小

这时我们切换回滚表空间，具体命令如下：

```
SQL> alter system set undo_tablespace=undotbs2 scope=both;
系统已更改。
```

脚本 3-26 切换回滚段的方法

现在观察，数据库当前的回滚表空间为 UNDOTBS2，如下：

```
SQL> show parameter undo
```

NAME	TYPE	VALUE
undo_management	string	AUTO
undo_retention	integer	900
undo_tablespace	string	UNDOTBS2

脚本 3-27 切换回滚段后，再查看当前回滚段是哪一个

如果我们要删除 UNDOTBS2 将要出错，提示正在使用中而无法删除，而原来的 UNDOTBS1 因为已经被切换为非当前 undo_tablespace 了，于是可以被删除了，如下：

```
SQL> drop tablespace undotbs2;
drop tablespace undotbs2
*
第 1 行出现错误:
ORA-30013: 还原表空间 'UNDOTBS2' 当前正在使用中
SQL> drop tablespace undotbs1 including contents and datafiles;
表空间已删除。
```

脚本 3-28 当前在用回滚段无法删除

通过试验证明，回滚表空间是真的可以新建多个，并且自由切换的，但是数据库当前使用的回滚表空间却只能有一个（注：RAC 数据库会有多个），这点请记住。曾祥同学，这个试验你看明白了吗？”

“明白了！”曾祥点头回答。

3.2.7.4 临时表空间新建与切换

“回滚表空间的特点是，数据库中 can 建立多个，但是目前的在用表空间却只能有一个。而临时表空间在数据库中也可以建多个，却可以被同时使用。

查看当前临时表空间有哪些，果然有两个，其中 TEMP_LJB 就是老师在描述“逻辑结构之 TABLESPACE”这一小节时新建的。

```
SQL> select tablespace_name,
       sum(bytes) / 1024 / 1024
       from dba_temp_files
       group by tablespace_name;
```

TABLESPACE_NAME	SUM(BYTES)/1024/1024
TEMP	59
TEMP_LJB	100

脚本 3-29 查看临时表空间大小

收获，不止 Oracle

而实际上当前的用户 ljb 早已是在用这个 TEMP_LJB 临时表空间了，因为我当初建 ljb 用户时是如下命令的：

```
create user ljb
identified by ljb
default tablespace tbs_ljb
temporary tablespace temp_ljb;
```

脚本 3-30 建用户时可指定表空间和临时表空间

该用户的临时表空间显然已经指定到 TEMP_LJB 下了。

```
SQL> select DEFAULT_TABLESPACE,TEMPORARY_TABLESPACE from dba_users where username='LJB';
DEFAULT_TABLESPACE      TEMPORARY_TABLESPACE
-----
TBS_LJB                 TEMP_LJB
```

脚本 3-31 查看用户的默认表空间和临时表空间

而其他用户显然还是在 TEMP 临时表空间下，因为我们并没有切换，我们随便举个 SYSTEM 用户看看便知：

```
SQL> select DEFAULT_TABLESPACE,
      TEMPORARY_TABLESPACE
      from dba_users
where username='SYSTEM';
DEFAULT_TABLESPACE      TEMPORARY_TABLESPACE
-----
SYSTEM                  TEMP
```

脚本 3-32 查看其他用户的临时表空间

我们可以用切换的命令让用户指定到不同的临时表空间去。我们做个试验如下：

```
sqlplus "/ as sysdba"
SQL> alter user system temporary tablespace TEMP_LJB;
用户已更改。
SQL> select DEFAULT_TABLESPACE,
      TEMPORARY_TABLESPACE
      from dba_users
      where username='SYSTEM';
DEFAULT_TABLESPACE      TEMPORARY_TABLESPACE
-----
SYSTEM                  TEMP_LJB
```

脚本 3-33 指定 SYSTEM 用户切换到指定的临时表空间

结果让人大吃一惊，我们居然把 SYSTEM 用户的临时表空间都改成 TEMP_LJB 了，不过这必须要是 SYS 用户方可，大家注意到我是用 sysdba 用户登录的。

不过我们继续观察发现当前数据库用户除了 2 个是用 TEMP_LJB，其他 26 个都是用默认的 TEMP 表空间。

```
select TEMPORARY_TABLESPACE,COUNT(*)
  from dba_users
 GROUP BY TEMPORARY_TABLESPACE;
```

TEMPORARY_TABLESPACE	COUNT(*)
TEMP	26
TEMP_LJB	2

脚本 3-34 观察不同用户在不同临时表空间的分配情况

如果想全部切换到 TEMP_LJB 临时表空间，那就要执行 26 次 alter user 的命令，有没有快捷的办法呢，其实是有的，如下：

```
SQL> alter database default temporary tablespace temp_ljb;
数据库已更改。
```

脚本 3-35 切换所有用户到指定临时表空间

这时我们观察数据库，发现所有的 28 个用户的默认临时表空间都已经为 TEMP_LJB 临时表空间，如下：

```
SQL> select TEMPORARY_TABLESPACE,COUNT(*)
  from dba_users
 GROUP BY TEMPORARY_TABLESPACE;
TEMPORARY_TABLESPACE      COUNT(*)
-----
TEMP_LJB                    28
```

脚本 3-36 所有用户默认临时表空间都被切换到 TEMP_LJB

看来临时表空间确实不同于回滚表空间，我们甚至可以建 10 个临时表空间，分别指定给 10 个不同的用户。

曾祥同学，临时表空间也说完了，你还有疑问吗？”

“没疑问了，谢谢梁老师。”

“很好，那我问问大家，回滚段表空间和临时表空间都可以建多个，这有什么好处呢？”

“我知道！”小莲等梁老师这个提问已经等很久了，在梁老师的引导下她已经将细心观察、探

收获，不止 Oracle

讨学习意义、知识如何应用变成了一种习惯，“回滚段建多个的目的是可以瘦身，原先的回滚段一直扩展导致空间浪费太多，新建出来的小一点，切换成功后删除原来旧的回滚表空间，磁盘空间就空余出来了。”看来小莲是认真观察过刚才梁老师试验中两个回滚段表空间的大小，清楚地记得 UNDOTBS1 已用到 5GB，新建的 UNDOTBS2 初始大小是 100MB，才有此结论。

“而临时表空间是为了避免竞争。”小莲接下来，说，“Oracle 可以为每个用户指定不同的临时表空间，每个临时表空间的数据文件都在磁盘的不同位置上，这样不是可以有效地避免 IO 竞争吗？”

“虽然不是很全面，但是说得非常好，同学们可以暂且这么理解。”老师忍不住鼓起掌来，老师心里明白，像小莲这样的同学，成为部门乃至公司的技术专家是迟早的事。

3.2.7.5 临时表空间组及其妙用

“Oracle 在设计上确实有很多过人之处。”梁老师继续说，“刚才大家看到 Oracle 可以为不同的用户指定不同的临时表空间从而减缓 IO 竞争，实际上在 Oracle 10g 以后推出的临时表空间组，可以做到为同一用户的不同 SESSION 设置不同的临时表空间，这可以说在缓解 IO 竞争方面再次迈出了大大的一步！”

为同一用户的不同 SESSION 设置不同临时表空间？还真能想啊，小莲感叹了一下。

“临时表空间组可以通过观察数据字典 dba_tablespace_groups 得到，如下说明数据库并没有设置临时表空间组：

```
SQL> select * from dba_tablespace_groups;
未选定行
```

脚本 3-37 查询临时表空间情况

实际上建临时表空间组很简单，只要新建一个临时表空间，然后加上 tablespace group tmp_grp1，就默认建成了一个名为 tmp_grp1 的临时表空间组了，比如我们新建 3 个临时表空间，并合并成一个临时表空间组，具体如下（注意，大家试验中路径要自行调整）：

```
SQL> create temporary tablespace temp1_1 tempfile 'E:\ORADATA\ORA10\DATAFILE\TMP1_1.DBF' size 100M
tablespace group tmp_grp1;
表空间已创建。
SQL> create temporary tablespace temp1_2 tempfile 'E:\ORADATA\ORA10\DATAFILE\TMP1_2.DBF' size 100M
tablespace group tmp_grp1;
表空间已创建。
SQL> create temporary tablespace temp1_3 tempfile 'E:\ORADATA\ORA10\DATAFILE\TMP1_3.DBF' size 100M
tablespace group tmp_grp1;
表空间已创建。
```

脚本 3-38 新建临时表空间组

现在我们观察数据字典 `dba_tablespace_groups`，发现已经可以查到记录了，原来建临时表空间组这么简单啊：

```
SQL> select * from dba_tablespace_groups;
GROUP_NAME          TABLESPACE_NAME
-----
TMP_GRP1            TEMP1_3
TMP_GRP1            TEMP1_2
TMP_GRP1            TEMP1_1
```

脚本 3-39 再查看临时表空间组情况，增加了 3 个成员

大家还记得之前我有建一个 `tmp_ljb` 的临时表空间吧，现在我想把这个临时表空间移到 `TMP_GRP1` 组里，具体操作如下：

```
SQL> alter tablespace temp_ljb tablespace group tmp_grp1;
表空间已更改。
```

脚本 3-40 可指定某临时表空间移到临时表空间组

继续观察，果然被加进临时表空间组里了：

```
SQL> select * from dba_tablespace_groups;
GROUP_NAME          TABLESPACE_NAME
-----
TMP_GRP1            TEMP_LJB
TMP_GRP1            TEMP1_3
TMP_GRP1            TEMP1_2
TMP_GRP1            TEMP1_1
```

脚本 3-41 移动临时表空间后，继续查看临时表空间组

现在我只要执行如下命令，就可以把 `ljb` 用户的默认临时表空间 `TEMP_LJB` 更改为临时表空间组 `TMP_GRP1`：

```
SQL> alter user LJB temporary tablespace tmp_grp1;
用户已更改。
```

脚本 3-42 将用户指定到临时表空间

查看发现用户默认临时表空间确实更改为默认临时表空间组了：

```
SQL> select temporary_tablespace
from dba_users
where username='LJB';
TEMPORARY_TABLESPACE
-----
```

TMP_GRP1

脚本 3-43 查看指定用户的临时表空间

这下 ljb 用户登录，在开启多个 SESSION 同时执行如下排序操作，命令为：

```
select a.table_name, b.table_name
  from all_tables a, all_tables b
 order by a.table_name;
```

脚本 3-44 该 SQL 执行会引发排序

虽然都是同一用户即 ljb 用户登录的，但是不同的 SESSION 都自动分配到了不同的临时表空间，以下是我们试验的一组结果，如下：

```
SQL> SELECT USERNAME,
        SESSION_NUM,
        TABLESPACE
  FROM V$SORT_USAGE;
USERNAME                SESSION_NUM  TABLESPACE
-----
LJB                      28          TEMP_LJB
LJB                      35          TEMP1_1
LJB                      38          TEMP1_2
LJB                      40          TEMP1_3
```

脚本 3-45 多进程执行大量排序的 SQL

而之前在未设置临时表空间组时，试验的结果是这样的，同一用户下的不同 SESSION 使用的临时表空间都是默认的 TEMP_LJB 临时表空间：

```
SQL> SELECT USERNAME,
        SESSION_NUM,
        TABLESPACE
  FROM V$SORT_USAGE;
USERNAME                SESSION_NUM  TABLESPACE
-----
LJB                      28          TEMP_LJB
LJB                      35          TEMP_LJB
LJB                      38          TEMP_LJB
LJB                      40          TEMP_LJB
```

脚本 3-46 查看排序后各 SESSION 使用的临时表空间情况

此外数据库中的临时表空间组也可以设置为多个，比如如下命令就是新增若干临时表空间

temp2_1、temp2_2、temp2_3，分配到新的 TMP_GRP2 临时表空间组中，然后分配给 YXL 用户。

```
create temporary tablespace temp2_1 tempfile 'E:\ORADATA\ORA10\DATAFILE\TMP2_1.DBF' size 100M
tablespace group tmp_grp2;
create temporary tablespace temp2_2 tempfile 'E:\ORADATA\ORA10\DATAFILE\TMP2_2.DBF' size 100M
tablespace group tmp_grp2;
create temporary tablespace temp2_3 tempfile 'E:\ORADATA\ORA10\DATAFILE\TMP2_3.DBF' size 100M
tablespace group tmp_grp2;
alter user YXL temporary tablespace tmp_grp2;
```

脚本 3-47 临时表空间组也可以设置多个

临时表空间组的推出，可以让我们往表空间组里不断新增临时表空间，让数据库在运行时自动从临时表空间组中选择各个临时表空间，不只是用户层面，而且是在 SESSION 层面进行 IO 均衡负载，极大地提升了数据库的性能。

大家觉得，Oracle 的这个设计思路是否值得我们去学习？”

“值得我们学习！”台下同学们齐声响亮地回答。

3.3 课程结束你给程序安上了翅膀

“至此，老师把体系逻辑结构的重点知识简要地介绍完毕了，同学们回想一下之前我给大家讲体系物理结构时，让大家探讨学习的意义，大家应该记忆犹新吧。同样，现在我也要问大家，听了我讲的前面几个小节后，觉得掌握这些知识有啥意义呢？”

或者我依然这么提问：知道这些体系逻辑结构知识和不知道这些知识的人在操作数据库时会有差异吗，你们有什么优势？

上次你们的回答可是让梁老师忏悔了许久哦同学们。”

回想起来的同学都会心地笑了，笑得最开心的是当时第一个起身说没意义的敬昱。

3.3.1 过度扩展与性能

“梁老师，您说过 extent 是 Oracle 数据库扩展的最小单位，而且大小是可以设置的。根据这两点我觉得如果某表（或者说某段）记录增长特别快，就可以考虑把这个 EXTENT 的大小设置得大一点，比如 initial extent 和 incremental extent 都设置比较大，这样申请扩展的次数就会减少，性能可以提高。”敬昱回答得很自信。

“说得太好了，我们来做一个试验吧，我们建两个表空间，一个是 TBS_LJB_A，一个是 TBS_LJB_B，命令如下：

收获，不止 Oracle

```
SQL> set timing on
SQL> create tablespace TBS_LJB_A
  2  datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB_A_01.DBF' size 1M
  3  autoextend on
  4  uniform size 64k;
```

表空间已创建。

已用时间: 00:00:00.44

```
SQL> create tablespace TBS_LJB_B
  2  datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB_B_01.DBF' size 2G ;
```

表空间已创建。

已用时间: 00:00:29.69

脚本 3-48 分别建统一尺寸和自动扩展的两个表空间

然后我们分别在两个表空间上建 T_A 和 T_B 表：

```
SQL> connect ljb/ljb
已连接。
SQL> set timing on
SQL> CREATE TABLE t_a (id int) tablespace TBS_LJB_A;
表已创建。
已用时间: 00:00:00.11
SQL> CREATE TABLE t_b (id int) tablespace TBS_LJB_B;
表已创建。
已用时间: 00:00:00.02
```

脚本 3-49 分别在两个不同表空间建表

接下来我们开始做试验，由于是插入 1 千万条，TBS_LJB_A 表空间的 1MB 大小很快就不够装了，不断自动扩大空间是可以预计的动作。而 TBS_LJB_B 表空间大小是固定的 2GB，足够容纳插入的 1 千万条记录。

下面我们来比较测试一下插入数据的速度：

```
SQL> insert into t_a select rownum from dual connect by level<=10000000;
已创建 10000000 行。
已用时间: 00:02:13.38
SQL> insert into t_b select rownum from dual connect by level<=10000000;
已创建 10000000 行。
已用时间: 00:00:21.09
```

脚本 3-50 分别比较插入的速度差异

一个是 2 分 13 秒，一个是 21 秒，为什么速度会差别如此之大呢？原来 T_A 段居然扩展了 1919 次，而 T_B 段才扩展了 86 次，这就是速度差异好几倍的原因。

```
SQL> select count(*) from user_extents where segment_name='T_A';
COUNT(*)
-----
1919
SQL> select count(*) from user_extents where segment_name='T_B';
COUNT(*)
-----
86
```

脚本 3-51 速度差异的原因

但是真正的原因其实是表空间在申请扩大空间时花费了大量时间，再继续看试验如下：

```
SQL> create tablespace TBS_LJB_C
2  datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB_C_01.DBF' size 2G
3  autoextend on
4  uniform size 64k;
```

表空间已创建。

已用时间: 00: 00: 28.75

```
SQL> connect ljb/ljb
```

已连接。

```
SQL> CREATE TABLE t_c (id int) tablespace TBS_LJB_C;
```

表已创建。

已用时间: 00: 00: 00.07

```
SQL> insert into t_c select rownum from dual connect by level<=10000000;
```

已创建 10000000 行。

已用时间: 00: 00: 22.79

脚本 3-52 表在 uniform 为 64K 的 tablespace 的插入情况

由于 TBS_LJB_C 表空间本身就够装下插入 1 千万行记录后的 T 表，表空间不需要扩展，速度也还不错，22 秒即完成，与前面 21 秒差别不大。

换句话说，就是如果老余的农场非常小，比如只有 10 平米，而猪圈就需要 100 平米，这时农场肯定要先扩大，然后才能让猪圈扩大，这时农场必须扩大到 100 平米，然后猪圈方可申请到 100 平米空间，这需要很大的开销，为啥呢？因为把空地改造成可以养殖的农场可不容易啊，比

收获，不止 Oracle

如要绿化，要装修等等。

如果农场已经扩展到 100 平米了，这时猪圈从 10 平米申请到 100 平米，即便每次申请的尺寸比较小，开销也不是太大。

其实这也比较好理解，对 Oracle 来说，表空间扩大是要格式化操作系统文件成为 Oracle 可以识别的数据库，这当然需要很大的开销了。如果空间已经足够大，其中的段申请 EXTENT，那都是可以识别的格式，无须格式化动作，当然开销就不是太大了，为什么？因为不需要绿化也不需要装修，这些开销大的动作之前已经完成了。

因此我们在建表空间时，需要预先规划好表空间的大小，如果段的扩展导致表空间不够而需要表空间去扩大，那开销是很大的，但是如果预先分配过多，也是一种浪费，需要我们根据实际应用去平衡。

现实中这样类似的案例也不少，这就是我们这次体系结构学习的意义之一了，希望大家认真听讲，有目的地去学习。”说到这里，梁老师停下来，等着别的同学继续发言。

3.3.2 PCTFREE 与性能

“梁老师，我们可以根据数据库对表更新的频繁程度，对表的 PCTFREE 做设置，免得产生行迁移，影响性能，这也算是我们学习的意义吧，我们可以根据自己的业务需求来调整 PCTFREE 参数。”小莲说。

“很好，下面我来做一组试验，大家可以模仿我的方式进行试验（其中 HR 用户是 Oracle 装库时自带的一个供大家参考的测试用户），具体如下：

SQL> DROP TABLE EMPLOYEES PURGE;
表已删除。
SQL> CREATE TABLE EMPLOYEES AS SELECT * FROM HR.EMPLOYEES ;
表已创建。
SQL> desc EMPLOYEES;

名称	是否为空?	类型
EMPLOYEE_ID		NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

脚本 3-53 PCTFREE 试验准备之建表

将几个字段扩大：

```
SQL> alter table EMPLOYEES modify FIRST_NAME VARCHAR2(2000);
表已更改。
SQL> alter table EMPLOYEES modify LAST_NAME  VARCHAR2(2000);
表已更改。
SQL> alter table EMPLOYEES modify EMAIL VARCHAR2(2000);
表已更改。
SQL> alter table EMPLOYEES modify PHONE_NUMBER  VARCHAR2(2000);
表已更改。
```

脚本 3-54 PCTFREE 试验准备之扩大字段

试验的关键步骤开始了，我们把如下几个从 20 个字节扩大为 2000 个字节的字段，填满数据，这必然将导致原先大量的行迁移产生，具体如下：

```
SQL> UPDATE EMPLOYEES
  2  SET FIRST_NAME = LPAD('1', 2000, '*'), LAST_NAME = LPAD('1', 2000, '*'), EMAIL = LPAD('1', 2000, '*'),
  3  PHONE_NUMBER = LPAD('1', 2000, '*');
已更新 107 行。
SQL> COMMIT;
提交完成。
```

脚本 3-55 PCTFREE 试验准备之更新表

现在我们来看看对这个表进行一个普通的查询，发现仅 107 行的记录居然产生 356 个逻辑读。（其中 select * from EMPLOYEES WHERE EMPLOYEE_ID=100;被反复执多次，取最终递归调用和物理读都为 0 时的结果比较公正）：

```
SQL> SET AUTOTRACE TRACEONLY
SQL> set linesize 1000
SQL> select * from EMPLOYEES;
执行计划
```

Plan hash value: 1445457117

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		107	52323	7 (0)	00:00:01
1	TABLE ACCESS FULL	EMPLOYEES	107	52323	7 (0)	00:00:01

Note

- dynamic sampling used for this statement

收获，不止 Oracle

统计信息

```
-----
0 recursive calls
0 db block gets
31 consistent gets
0 physical reads
0 redo size
5447 bytes sent via SQL*Net to client
477 bytes received via SQL*Net from client
9 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
107 rows processed
```

脚本 3-56 PCTFREE 试验准备之查看逻辑读情况

下面我们用 EMPLOYEES 新建一个 EMPLOYEES_BK 表，这等于消除了行迁移，发现逻辑读减少了许多，从 31 缩减为 20（其中 select * from EMPLOYEES_BK;也是执行过 2 次以上的结果，保证公正）：

```
SQL> CREATE TABLE EMPLOYEES_BK AS select * from EMPLOYEES;
```

表已创建。

```
SQL> SET AUTOTRACE TRACEONLY
```

```
SQL> set linesize 1000
```

```
SQL> select * from EMPLOYEES_BK;
```

执行计划

```
-----
Plan hash value: 2676497765
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		107	52323	6 (0)	00:00:01
1	TABLE ACCESS FULL	EMPLOYEES_BK	107	52323	6 (0)	00:00:01

Note

```
-----
- dynamic sampling used for this statement
```

统计信息

```
-----
0 recursive calls
0 db block gets
20 consistent gets
0 physical reads
```

```

0 redo size
5447 bytes sent via SQL*Net to client
477 bytes received via SQL*Net from client
9 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
107 rows processed

```

脚本 3-57 PCTFREE 试验准备之消除行迁移后的逻辑读情况

同学们，这里其实我是做了一个极端的试验，原先填充 20 个字节的长度被更新为 200 个字节的长度，好比小不点忽然变成超级大胖子，行迁移是必然的，而且是大批量。我这里才 100 多条记录的小表，逻辑读居然就差别如此之大！

另外大家看到了，**其实消除行迁移的一个简单方法，就是数据重建**，这个 EMPLOYEES_BK 表就没有行迁移现象，所以逻辑读少多了！

另外这里是默认取值，EMPLOYEES 表的 PCTFREE 默认为 10，查询数据库，果然是如此：

```

SQL> select pct_free from user_tables where table_name='EMPLOYEES';
PCT_FREE
-----
10

```

脚本 3-58 查看 EMPLOYEES 的 PCTREE 值

我们可以自行调整这个值，命令为：

```

SQL> alter table EMPLOYEES pctfree 20 ;
Table altered
SQL> select pct_free from user_tables where table_name='EMPLOYEES';
PCT_FREE
-----
20

```

脚本 3-59 调整 PCTFREE 的方法

这个参数到底该设置多大合理，是需要深入了解和测试的，并不简单，实际工作中也有不少调整这个参数优化数据库系统的案例，并取得了不错的效果！”

3.3.3 行迁移与优化

“梁老师，如何发现表存在大量行迁移？”小莲忽然插话了。

“很好，老师正要说这个，方法大家可以看下面的说明：

收获，不止 Oracle

--首先建 chained_rows 相关表，这是必需的步骤

```
sqlplus "/ as sysdba"
```

```
SQL> @?/rdbms/admin/utlchain.sql
```

表已创建。

----以下命令针对 EMPLOYEES 表和 EMPLOYEES_BK 做分析，将产生行迁移的记录插入到 chained_rows 表中

```
SQL> analyze table EMPLOYEES list chained rows into chained_rows;
```

Table analyzed

```
SQL> analyze table EMPLOYEES_BK list chained rows into chained_rows;
```

Table analyzed

--然后分析可以得知，目前假如访问 EMPLOYEES 表，将会有 180 个产生行迁移的动作，而该表记录不过才 108 条。

```
SQL> select count(*) from chained_rows where table_name='EMPLOYEES';
```

```
COUNT(*)
```

```
-----  
180
```

```
SQL> select count(*) from chained_rows where table_name='EMPLOYEES_BK';
```

```
COUNT(*)
```

```
-----  
0
```

脚本 3-60 发现存在行迁移的方法

根据这个方法，我们就可以了解到数据库表中有哪些存在行迁移严重的情况，并做适当的改进，比如重建新表消除行迁移，然后对 PCTFREE 做适当的调整，等等。如果对当前用户的所有表做分析，可以考虑用如下方法：

```
select 'analyze table '|| table_name ||' list chained rows into chained_rows;' from user_tables;
```

```
select * from chained_rows;
```

脚本 3-61 检查所有表是否存在行迁移的脚本

大家学会这个后，有没有一种要去自己的数据库环境试验一下的冲动？”梁老师笑着问。

“有！”同学们回答得很大声。

“不过，老师还是那句话，只要是生产环境，就一定要谨慎操作，所有在业务高峰期做的操作都是危险操作，你们可以在凌晨系统无人使用时，适当检查一下自己怀疑产生很多行迁移导致 IO 增加的表，切记安全第一！”梁老师不断强调安全。

3.3.4 块的大小与应用

“BLOCK 除了谈这个 PCTFREE 属性外，还有本身设置多大的问题。”梁老师微笑着说，

“BLOCK 设置多大还是很有技巧的，为什么有的系统设置 8KB 甚至 4KB、2KB，而有的系统设置 16KB 甚至 32KB 大呢？现在我才和你们讨论这个问题，如果今天我上体系逻辑结构课，大家连 BLOCK 是什么都不知道，就无从讨论，更不可能根据业务场景来规划数据库块大小的设计了。

我之前说过，BLOCK 是 Oracle 最小的单位。如果 Oracle 是单块读，则一次读取一个块，就是一个 IO，当然如果是一次读取多个块，那还是算一个 IO，这称之为多块读。这里有一个问题，如果块越大，装的行记录就越多，那所需要的块就越少，换句话说，读取记录产生的 IO 就越少，因此 BLOCK 设置可以理解为越大越好，是吗，大家觉得对吗？”

大家都沉默了，估计是一时没想明白。

“实际情况是，对于数据仓库 OLAP 的数据库应用，我们一般倾向于 BLOCK 尽量大，而 OLTP 应用，一般倾向于 BLOCK 尽量不要太大。OLAP 和 OLTP 的差别在于，前者一般查询返回大量的数据，而后者查询返回极少量数据。前者一般用户不多，并发不大，后者一般用户很多，并发很大。因此 OLAP 系统最多的查询方式应该是全表扫描，而 OLTP 系统最多的方式应该是索引读。这其中的原理并不难，具体细节我们会在后续优化的课程中有详细描述。下面我演示一些具体的例子，大家仔细观察揣摩我说过的话。

我之前建过一个名为 TBS_LJB_16k 的 100MB 的表空间，块大小是 16KB，现在我们就分别在 TBS_LJB 和 TBS_LJB_16K 的表空间上做查询，看看有何差异。

以下是重建这两个表空间的命令，尺寸都分别为 1GB。具体命令如下：

```
drop tablespace TBS_LJB INCLUDING CONTENTS AND DATAFILES;
create tablespace TBS_LJB
datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB_01.DBF' size 1G;

drop tablespace TBS_LJB_16K INCLUDING CONTENTS AND DATAFILES;
create tablespace TBS_LJB_16K
blocksize 16K
datafile 'E:\ORADATA\ORA10\DATAFILE\TBS_LJB_16k_01.DBF' size 1G;
```

脚本 3-62 块的大小应用环境工作（分别建 8K 和 16K 的表空间）

接下来在两个表空间分别构造两张记录达到 300 多万的大表，并建索引。在 BLOCK 为 16KB 的 TBS_LJB_16K 表空间完成 T_16K 这个表的准备工作，具体如下：

```
drop table t_16k purge;
create table t_16k tablespace tbs_ljb_16k as select * from dba_objects ;
insert into t_16k select * from t_16k;
insert into t_16k select * from t_16k;
insert into t_16k select * from t_16k;
insert into t_16k select * from t_16k;
insert into t_16k select * from t_16k;
```

收获，不止 Oracle

```
insert into t_16k select * from t_16k;
commit;
update t_16k set object_id=rownum ;
commit;
create index idx_object_id on t_16k(object_id);
```

脚本 3-63 块的大小应用准备工作（在 16K 表空间建表）

接下来在 BLOCK 为 8KB 的 TBS_LJB 表空间中构造环境，完成大表 T_8K 的准备工作，具体如下：

```
drop table t_8k purge;
create table t_8k tablespace tbs_ljb as select * from dba_objects ;
insert into t_8k select * from t_8k;
insert into t_8k select * from t_8k;
insert into t_8k select * from t_8k;
insert into t_8k select * from t_8k;
insert into t_8k select * from t_8k;
insert into t_8k select * from t_8k;
insert into t_8k select * from t_8k;
commit;
update t_8k set object_id=rownum ;
commit;
create index idx_object_id_8k on t_8k(object_id);
```

脚本 3-64 块的大小应用准备工作（在 8K 表空间建表）

现在可以方便地进行测试了，首先是观察 `select count(*) from t_16K` 的性能：

```
SQL> set autotrace on
SQL> set linesize 1000
SQL> set timing on
SQL> select count(*) from t_16k;
COUNT(*)
```

3558784

已用时间: 00: 00: 04.48

执行计划

Plan hash value: 3599734656

Id	Operation	Name	Rows	Cost (%CPU)	Time
0	SELECT STATEMENT		1	6661 (1)	00:01:20
1	SORT AGGREGATE		1		

```
| 2 | TABLE ACCESS FULL | T_16K | 3892K | 6661 (1) | 00:01:20 |
```

Note

- dynamic sampling used for this statement

统计信息

```

0 recursive calls
0 db block gets
24204 consistent gets
19554 physical reads
0 redo size
415 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 3-65 BLOCK 为 16K 的表空间全表扫描性能

接下来观察 select count(*) from t_8K 的性能情况：

```
SQL> select count(*) from t_8k;
```

```
COUNT(*)
```

```
3558848
```

```
已用时间: 00: 00: 04.94
```

执行计划

```
Plan hash value: 576579961
```

Id	Operation	Name	Rows	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10731 (1)	00:02:09
1	SORT AGGREGATE		1		
2	TABLE ACCESS FULL	T_8K	2712K	10731 (1)	00:02:09

Note

- dynamic sampling used for this statement

统计信息

```
0 recursive calls
```


收获，不止 Oracle

0	db block gets
48871	consistent gets
43192	physical reads
0	redo size
415	bytes sent via SQL*Net to client
400	bytes received via SQL*Net from client
2	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
1	rows processed

脚本 3-66 BLOCK 为 8K 的表空间的全表扫描性能

比较发现，BLOCK 为 16KB 的 TBS_LJB_16K 表空间环境下查询需要 1 分 20 秒，而 BLOCK 为 8KB 的环境需要 2 分 9 秒，逻辑读也差别很大，由此可见，在类似 OLAP 系统的环境下，块设置越大性能越会有提升。

但是在索引读的环境下，就没有什么性能优势了，比较下面两个语句的查询，性能上基本上无差异。

BLOCK 为 8KB 的环境下，执行 `select * from t_8k where object_id=29`；如下：

```
SQL> select * from t_8k where object_id=29;
```

执行计划

Plan hash value: 3863279139

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	177	4 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	T_8K	1	177	4 (0)	00:00:01
* 2	INDEX RANGE SCAN	IDX_OBJECT_ID_8K	1		3 (0)	00:00:01

Predicate Information (identified by operation id):

2 - access("OBJECT_ID">=29)

Note

- dynamic sampling used for this statement

统计信息

0	recursive calls
0	db block gets
5	consistent gets

```

0    physical reads
0    redo size
1199 bytes sent via SQL*Net to client
400  bytes received via SQL*Net from client
2    SQL*Net roundtrips to/from client
0    sorts (memory)
0    sorts (disk)
rows processed

```

脚本 3-67 BLOCK 大小为 8K 的表空间的索引读性能

BLOCK 为 16KB 的环境下，执行 `select * from t_16k where object_id=29`;如下：

```
SQL> select * from t_16k where object_id=29;
```

执行计划

```
-----
Plan hash value: 3672296614
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	94	4 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	T_16K	1	94	4 (0)	00:00:01
* 2	INDEX RANGE SCAN	IDX_OBJECT_ID	1		3 (0)	00:00:01

```
-----
Predicate Information (identified by operation id):
```

```
-----
2 - access("OBJECT_ID"=29)
```

统计信息

```

-----
0    recursive calls
0    db block gets
5    consistent gets
0    physical reads
0    redo size
1199 bytes sent via SQL*Net to client
400  bytes received via SQL*Net from client
2    SQL*Net roundtrips to/from client
0    sorts (memory)
0    sorts (disk)
1    rows processed

```

脚本 3-68 BLOCK 大小为 16K 的表空间的索引读性能

比较发现，索引读返回少量记录这样的 OLTP 主打环境下，块的大小对性能影响不大。那为什么 OLTP 系统倾向于让块的尺寸小一些呢？主要是因为如果块太大，容易导致大量并发查询及更新操作都指向同一个数据块，从而产生热点块竞争。”

梁老师说到这里，小莲有些恍然大悟了，看来体系结构认真学习是很有必要的，梁老师无论在描述体系结构的物理结构还是逻辑结构时，都带来了很生动的关于学习意义的描述，而这些显然是和实战息息相关的技能，小莲觉得自己受益颇多。

“今天的体系逻辑结构就说到这里，大家回去好好思考一下都学了什么，并把梁老师的试验都操作一遍。另外别忘了复习之前的第一课堂学到的学习路线图与体系物理结构，这些知识是后续学习的重要基础。

大家知道我的风格，梁老师并没有把所有的细节知识说完，重点学习探讨的是二八现象中的‘二’，而非全部知识。我们的课堂更多的是传授学习思想和方法，课堂期间的思维启迪、经验交流和案例分享应该是网上和官方文档不会提及的，应该说这些比知识本身更重要。

接下来我会给大家描述的是索引的结构。大家是否觉得我上过的课堂比较注重实战？而这个索引课堂更是注重实战，相信不少同学学完后，都会跃跃欲试想立即登录自己的数据库环境一试身手。

下周同一时刻，我们这里再会，敬请期待……”

梁老师说到这里，宣布今天的课程结束。

同学们有些依依不舍，同时心里又期待万分。

第 4 章



祝贺，表的设计成就英雄

4.1 表的设计之五朵金花

表设计的“五朵金花”如图 4-1 所示。

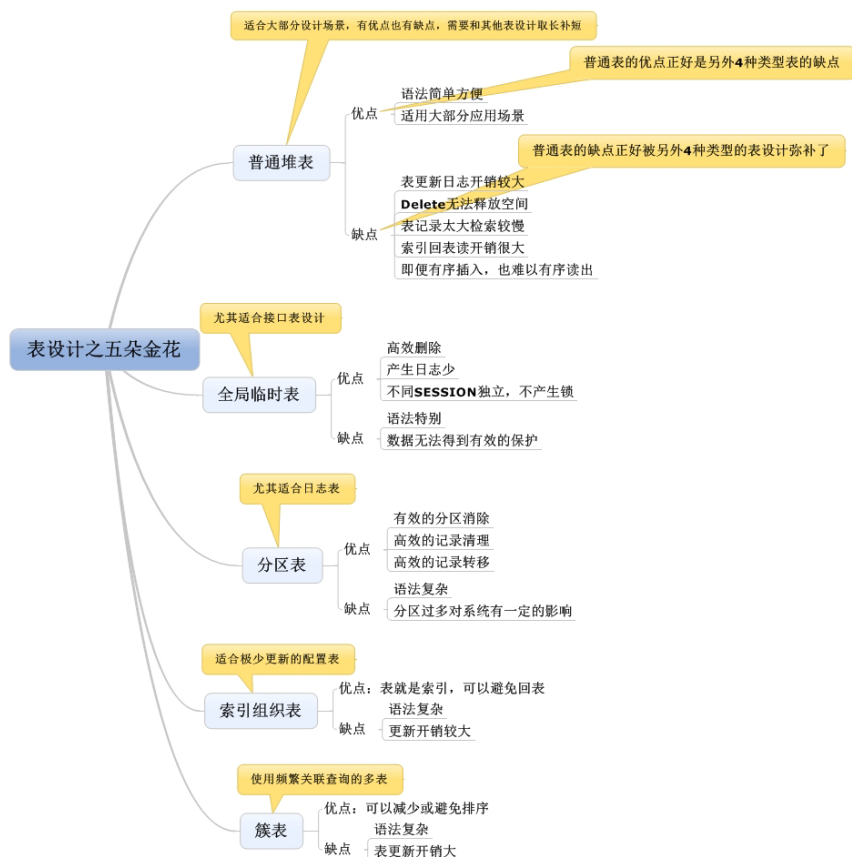


图 4-1 表设计优缺点之五朵金花

4.2 表的特性从老余一家展开描述

4.2.1 老余一家各施所长

“在前面的课程中，大家对既开过店又开过农场的勤劳的生意人老余一家应该都很熟悉了吧，现在老师再给大家说一个他们一家生活中的小故事。

话说有一次老余的店要进一些新衣服，他们一家三口是这么规划的：

- ① 老余业务熟悉，主要负责联系货源厂家、商谈价格。
- ② 小余年轻力壮，主要负责完成出门提货、搬运、货物上架等工作。
- ③ 余太太认真心细，主要负责比对审核成本，检查货物质量等工作。

然后，我的故事说完了。”

“不是吧，这么短！”同学们都乐了，少数故意在大声抗议。

“就这么短啊，老师只是想和大家说，每个人都有每个人的特点和优势，要善于发掘和利用，才可以把事情做好。而我们所学课程的知识点其实也是如此，所学知识点，有的适用于这个应用场景，却不适合另外一个场景，要学会选择性地使用技术。好比让年纪大的老余和余太太去奔波搬运，显然不合适。让不熟悉业务的小余去联系厂家商谈价格，也不合适。

老余一家各施所长是正确的选择，我们接下来的表设计的学习就会让大家深刻理解和，其实学习和生活是一样的，要学会选择适用场景。

技术其实并不难，最难的是如何选择。”

4.2.2 普通堆表不足之处

普通堆表的不足之处如图 4-2 所示。

“同学们好，今天我们开始探讨表设计的相关技术，表位于体系物理结构的数据文件部分，在体系结构逻辑结构中，一张表就是一个段，如果该表有索引，一个索引就是一个段。这些知识，大家还清楚吧？”老师问。

“清楚！”大家都回答得很干脆。

“可能大家会觉得表没什么可学的，太简单了，建表的语法也超简单，即便谁忘记了，随便上网一搜，也是一大片，无非就是 CREATE TABLE 某表 (列 1 类型 1, 列 2 类型 2……)等诸如此类的。

实际上，Oracle 表的分类是多种多样的，除了普通表外，还有全局临时表、外部表、分区表、索引组织表等等具有其他特性的表。虽然普通表基本上可以实现所有的功能，但是这是说功能，而不是说性能。如果我们善于在合适的场合选择合适的技术，这些“特殊”的表往往能在系统应用设计的性能方面，发挥出巨大的作用。

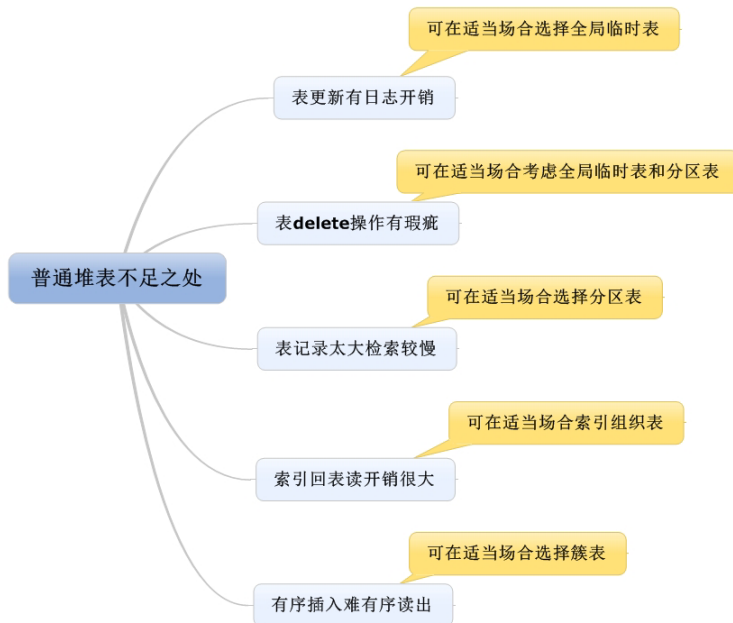


图 4-2 普通堆表不足之处

其实表设计的这整个章节，我们主要就是在强调什么场合该选择什么技术，而这句话本身就告诉我们，没有最高级的技术，只有最适合的技术。

结束本章节后，大家就会更加深刻地体会到，各种类型表都有优缺点，我们要善于取长补短，灵活利用，我们本着挑剔的态度来探讨普通表的缺点，至于优点，我们会在本次课程的最后阶段进行综合探讨。

4.2.2.1 表更新日志开销较大

老师先准备了分析数据库产生多少日志的脚本，如下：

```
select a.name,b.value
  from v$statname a,v$mystat b
 where a.statistic#=b.statistic#
 and a.name='redo size';
```

脚本 4-1 查看产生多少日志

该脚本是利用 v\$statname 和 v\$mystat 两个动态性能视图来跟踪当前 SESSION 操作产生的日志量，使用方法很简单：首次先执行该脚本，查看日志大小，随即执行你的更新语句，再执行该脚本返回的日志大小，两者相减，就是你此次更新语句产生的日志大小，我们做试验如下。

收获，不止 Oracle

```
sqlplus "/ as sysdba"
--其中该视图需要先 sqlplus /as sysdba 登录授权如下后方可执行
SQL> grant all on v_$mystat to ljb;
授权成功。
SQL> grant all on v_$statname to ljb;
授权成功。
connect ljb/ljb
SQL> drop table t purge;
表已删除。
SQL> create table t as select * from dba_objects ;
表已创建。
--以下创建视图，方便后续直接用 select * from v_redo_size 进行查询
SQL> create or replace view v_redo_size as
    select a.name,b.value
    from v_$statname a,v_$mystat b
    where a.statistic#=b.statistic#
    and a.name='redo size';
视图已创建。
```

脚本 4-2 试验准备工作，建观察 redo 的视图

```
SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           110132
SQL> delete from t ;
已删除 55708 行。
SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           20233304
```

脚本 4-3 观察删除记录产生多少 redo

好了，大家看看，这个试验说明了什么？”老师问。

“说明删除语句产生了 **20233304-110132=20123172** 的日志量。”小莲回答得很迅速。

“回答得很好，这个量的单位是字节数，因此大致是产生了 20MB 的日志。下面我们继续上述的试验，我们完成一个插入语句，如下：

```
SQL> insert into t select * from dba_objects;
已创建 55708 行
SQL> select * from v_redo_size;
```

NAME	VALUE

redo size	26358732

脚本 4-4 观察插入记录产生多少 redo

大家又有什么发现呢？”老师继续问。

“插入语句产生了 $26358732 - 20233304 = 6125428$ 也就是大约 6MB 的数据。”还是小莲抢答到了。

“完成了删除和插入后，我们试验一下修改动作，如下：

```
SQL> update t set object_id=rownum;
已更新 55708 行。
SQL> select * from v_redo_size;
```

NAME	VALUE

redo size	33335524

脚本 4-5 观察更新记录产生多少 redo

好了，大家看看，这个试验说明了什么？”老师问。

“更新语句产生了 $33335524 - 26358732 = 6976792$ 大约 7MB 的数据”。这回被曾祥抢答到了。

“很好，老师的这三个试验说明了对表的更新操作，无论是删除、插入还是修改，都会产生日志。大家在前面学过体系结构，应该很清楚地知道这些日志是用于数据库的备份和恢复的。如果仅从性能角度而不从安全性角度来考虑，更新表写日志就意味着数据库多做了额外的事情而影响了效率，虽说安全第一，不过在某些特定的场合，某些表的记录只是作为中间结果临时运算而根本无须永久保留，这些表无须写日志，那就既高效又安全了！

这就是后续老师将会说的全局临时表，敬请期待。”老师笑着说。

“此外大家还记得学习体系结构时小莲同学关于插入、删除、更新三种语句产生 UNDO 量的排名的回答吗？老师没有正面回答她的答案是否正确，这个悬念到现在大家可以自行查看结果了，大家看看是什么情况？”老师问。

“我的回答是错的，删除产生的 REDO 最多！”小莲仔细看了看大屏幕后有些无奈地回答。

“为什么？”老师问。

“我明白了，其实删除产生的 UNDO 最多，而 UNDO 也是需要 REDO 保护的，所以虽然本身产生的 REDO 不多，但是由于删除时的 UNDO 量最大，用于保护 UNDO 的 REDO 量也最大，所以加在一起，删除产生的 REDO 也就可能最多了。”小莲想了一小会儿后，有些恍然大悟。

“很好，我就喜欢大家自己找到答案。”老师笑着肯定了小莲的回答。

4.2.2.2 delete 无法释放空间

“同学们都知道 delete 操作，删除表记录的命令，再简单不过了。可是大家不知道，实际工作中不少性能问题都和 delete 操作有关，大家知道是什么原因吗？”老师问。

“因为 delete 是最耗性能的操作，产生的 UNDO 最多，而且因为 UNDO 需要 REDO 来保护的缘故，delete 产生的 REDO 也是最大，所以不少性能问题都和 delete 操作有关。”晶晶自信满满地回答。

“回答得不错，老师再做一个试验，可能结果会让大家意想不到。

```
SQL> drop table t purge;
```

表已删除。

```
SQL> create table t as select * from dba_objects ;
```

表已创建。

```
SQL> set autotrace on
```

```
SQL> select count(*) from t;
```

COUNT(*)

55709

执行计划

Plan hash value: 2966233522

Id	Operation	Name	Rows	Cost (%CPU)	Time
0	SELECT STATEMENT		1	175 (1)	00:00:03
1	SORT AGGREGATE		1		
2	TABLE ACCESS FULL	T	54422	175 (1)	00:00:03

Note

- dynamic sampling used for this statement

统计信息

```
0 recursive calls
0 db block gets
771 consistent gets
0 physical reads
0 redo size
414 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
```

```

0  sorts (disk)
1  rows processed

```

脚本 4-6 观察未删除表时产生的逻辑读

现在我执行 `delete from t` 命令后，再 `select count(*) from t` 全扫描该表，逻辑读应该会大幅度减少吧？”老师问。

“是！”台下基本上都是一致的回答。

“好，我们继续试验如下：

```

SQL> set autotrace off
SQL> delete from t ;
已删除 55709 行。
SQL> commit;
提交完成。
SQL> set autotrace on
SQL> select count(*) from t;
COUNT(*)
-----
0
执行计划
-----
Plan hash value: 2966233522
-----
| Id | Operation          | Name | Rows | Cost (%CPU) | Time      |
-----
| 0 | SELECT STATEMENT   |      | 1 | 174 (0) | 00:00:03 |
| 1 | SORT AGGREGATE     |      | 1 |          |          |
| 2 | TABLE ACCESS FULL| T     | 1 | 174 (0) | 00:00:03 |
-----

Note
-----
- dynamic sampling used for this statement
统计信息
-----
0 recursive calls
0 db block gets
771 consistent gets
0 physical reads
0 redo size
411 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)

```

收获，不止 Oracle

0	sorts (disk)
1	rows processed

脚本 4-7 观察 delete 删除 t 表所有记录后，居然逻辑读不变

什么情况啊？同学们。”老师问。

“好奇怪啊，记录数从 55709 减少到 0 条记录了，怎么逻辑读还是 771 个啊！”小莲忍不住喊出声来。

“很奇怪啊，那继续观察下面试验：

```
SQL> truncate table t;
表被截断。
SQL> select count(*) from t;
COUNT(*)
-----
0
执行计划
-----
Plan hash value: 2966233522
-----
| Id | Operation                | Name | Rows  | Cost (%CPU)  | Time          |
-----
| 0 | SELECT STATEMENT         |      | 1     | 2 (0)        | 00:00:01     |
| 1 | SORT AGGREGATE           |      | 1     |              |              |
| 2 | TABLE ACCESS FULL       | T    | 1     | 2 (0)        | 00:00:01     |
-----
Note
-----
- dynamic sampling used for this statement
统计信息
-----
0 recursive calls
0 db block gets
3 consistent gets
0 physical reads
0 redo size
411 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 4-8 truncate 清空表后，逻辑读终于大幅度下降了

现在是什么情况呢？”老师问。

“老师，逻辑读从 771 减少到 3 个了，代价也从 170 多减少到 2 个了，真是神奇啊。”曾祥回答的语调也显得很惊讶。

“看来大家观察都很仔细，这里很显然看出，`delete` 删除并不能释放空间，虽然 `delete` 将很多块的记录删除了，但是空块依然保留，Oracle 在查询时依然会去查询这些空块。而 `truncate` 是一种释放高水平位的动作，这些空块被回收，空间也就释放了。

举个简单的例子，好比我来到 XX 大楼统计里面的人数，我从 1 楼找到 20 楼，每层的房间都打开去检查了一下，发现实际情况是一个人都没有。我很后悔自己累得半死却得出没人的结论，但问题是，你不打开房间，怎么知道没人呢，这就类似 `delete` 后空块的情况。而与 `truncate` 有些类似的生动例子就是，我想统计 XX 大楼里的人数，结果发现，XX 大楼被铲平了，啥房间都没有了，于是我飞快地得出结论，XX 大楼里没有人。

听明白了吗？”老师问。

老师的比喻还真是生动，大家终于明白了前面的试验结果为什么这么奇怪了。

“不过 `truncate` 显然不能替代 `delete`，因为 `truncate` 是一种 DDL 操作而非 DML 操作，`truncate` 后面是不能带条件的，`truncate table t where...` 是不允许的。但是如果表中这些 `where` 条件能形成有效的分区，Oracle 是支持在分区表中做 `truncate` 分区的，命令大致为 `alter table t truncate partition '分区名'`，如果 `where` 条件就是分区条件，那等同于换角度实现了 `truncate table t where...` 的功能。

这就是分区表最实用的功能之一了，高效地清理数据，释放空间，老师将在后续的章节中详细描述分区表的特性，敬请期待。老师说话的表情好像要为自己做广告一样，引来了同学们阵阵大笑。

“此外同学们要注意的是，当大量 `delete` 删除再大量 `insert` 插入时，Oracle 会去这些 `delete` 的空块中首先完成插入（直接路径插入除外），所以频繁 `delete` 又频繁 `insert` 的应用，是不会出现空块过多的情况的，这点大家还是要知道一下，不能草木皆兵。”老师又补充了一点。

4.2.2.3 表记录太大检索较慢

“刚才已经描述普通表操作的两点瑕疵，一是在无须考虑备份，允许不产生日志时普通表操作依然会产生大量日志，无法节省开销；二是 `delete` 操作开销大且无法释放空间。接下来老师要和大家探讨一下大表检索的相关技巧。

大家知道，一张表其实就是一个 SEGMENT，一般情况下我们都需要遍历该 SEGMENT 的所有 BLOCK 来完成对该表进行更新查询等操作，在这种情况下，表越大，更新查询操作就越慢！

有没有什么好方法能提升检索的速度呢？主要思路就是缩短访问路径来完成同样的更新查询操作，简单地说就是完成同样的需求访问 BLOCK 的个数越少越好。Oracle 为了尽可能减少访问路径提供了两种主要技术，一种是索引技术，另一种则是分区技术。接下来我们以 `select * from`

收获，不止 Oracle

t where created>= xxx and created <=xxx 这个简单的 SQL 语句为例进行分析。

首先说索引，这是 Oracle 中最重要也是最实用的技术之一，在本例中，如果 created>= xxx and created <=xxx 返回的记录非常少，或者说和 T 表的总记录相比非常少，则在 created 列建索引能极大提升该语句的效率。比如我们建成了一个 idx_t 的索引，在该 SQL 查询时我们首先会访问 idx_t 这个新建出来的索引段，然后通过索引段和表段的映射关系，迅速从表中获取行列的信息并返回结果。具体技术细节老师将会在后续的索引章节中做详细的描述，大家要记得，索引本身也是一把双刃剑，既能给数据库开发应用带来极大的帮助，也会给数据库带来不小的灾难。

减少访问路径的第二种技术就是分区技术了，我们把普通表 T 表改造为分区表，比如以 created 这个时间列为分区字段，比如从 2010 年 1 月到 2012 年 12 月按月建 36 个分区。早先的 T 表就一个 T 段，现在情况变化了，从 1 个大段分解成了 36 个小段，分别存储了 2010 年 1 月到 2012 年 12 月的信息，此时假如 created>= xxx and created <=xxx 这个时间跨度正好是落在 2012 年 11 月，那 Oracle 的检索就只要完成一个小段的遍历即可，假设这 36 个小段比较均匀，我们就可以大致理解为访问量只有原来的三十六分之一，大幅度减少了访问路径，从而高效地提升了性能。

这就是分区表了，除了之前描述的高效清理数据外，还有减少访问路径的神奇本领，是不是越来越值得大家期待了？”老师问。

“期待！”曾祥声音大得离谱，他总是这样似捣乱非捣乱的样子。

“看来普通表的不足之处还真是不少，接下来我们继续看看普通表操作中还有哪些不足之处。”老师喝了口水，继续领着大家一起探索。

4.2.2.4 索引回表读开销很大

“我们来查看这个例子，大家注意看老师的这个试验，具体如下：

```
SQL> drop table t purge;
```

表已删除。

```
SQL> create table t as select * from dba_objects where rownum<=200;
```

表已创建。

```
SQL> create index idx_obj_id on t(object_id);
```

索引已创建。

```
SQL> set linesize 1000
```

```
SQL> set autotrace traceonly
```

```
SQL> select * from t where object_id<=10;
```

已选择 9 行。

执行计划

Plan hash value: 134201588

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	
----	-----------	------	------	-------	-------------	------	--

```

-----
| 0 | SELECT STATEMENT |          | 9 | 1593 | 2 (0)| 00:00:01 |
| 1 |  TABLE ACCESS BY INDEX ROWID | T        | 9 | 1593 | 2 (0)| 00:00:01 |
|* 2 |  INDEX RANGE SCAN   | IDX_OBJ_ID | 9 |      | 1 (0)| 00:00:01 |
-----

```

Predicate Information (identified by operation id):

```

-----
2 - access("OBJECT_ID"<=10)

```

Note

```

-----
- dynamic sampling used for this statement

```

统计信息

```

-----
0 recursive calls
0 db block gets
4 consistent gets
0 physical reads
0 redo size
1456 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
9 rows processed

```

脚本 4-9 观察 TABLE ACCESS BY INDEX ROWID 产生的开销

这里大家注意一个地方，TABLE ACCESS BY INDEX ROWID，大家看到没？”老师问。

“看到了！”大家很快就找到了。

“一般来说，根据索引来检索记录，会有一个先从索引中找到记录，再根据索引列上的 ROWID 定位到表中从而返回索引列以外的其他列的动作，这就是 TABLE ACCESS BY INDEX ROWID，这在后续索引章节将会有详细描述。

下面老师再做一个试验，大家自己认真观察看看有什么不同之处，如下：

```
SQL> select object_id from t where object_id<=10;
```

已选择 9 行。

执行计划

```
-----
Plan hash value: 188501954

```

```
-----
| Id | Operation          | Name      | Rows  | Bytes | Cost (%CPU) | Time      |
-----

```

收获，不止 Oracle

```
| 0 | SELECT STATEMENT |          | 9 | 117 | 1 (0) | 00:00:01 |
|* 1 | INDEX RANGE SCAN | IDX_OBJ_ID | 9 | 117 | 1 (0) | 00:00:01 |
```

Predicate Information (identified by operation id):

1 - access("OBJECT_ID"<=10)

Note

- dynamic sampling used for this statement

统计信息

```
0 recursive calls
0 db block gets
2 consistent gets
0 physical reads
0 redo size
508 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
9 rows processed
```

脚本 4-10 观察如果消除 TABLE ACCESS BY INDEX ROWID 的开销情况

有什么新发现啊？”老师问。

“老师，没有 TABLE ACCESS BY INDEX ROWID 了！”曾祥最先回答了。

“为什么没了？”老师问。

“因为语句从 `select * from t where object_id<=10` 改写为 `select object_id from t where object_id<=10` 了，不用从索引中回到表中获取索引列以外的其他列了。”曾祥愣了半天没回答上来，被小莲给抢答了。

“那性能有提升吗？”老师继续问。

“有啊，逻辑读从 4 变为 2，代价从 2 变为 1。”小莲早就观察到了，回答得非常迅速。

“很好，避免回表从而使性能提升这是一个很简单的道理，少做事性能当然提升了。只是 `select * from t` 和 `select object_id from t` 毕竟不等价，有没有什么方法可以实现写法依然是 `select * from t`，但是还是可以不回表呢？”老师问。

大家有些发懵，一时没人回答。

“大家答不上来正常，普通表是做不到的，能实现这种功能的只有索引组织表，老师将会在后续章节中介绍给大家，敬请期待。”老师的这些广告式的口吻引发了大家的笑声，听起来像是

老师做的第4个广告了。

4.2.2.5 有序插入却难有序读出

“大家跟老师再做一组试验。在对普通表的操作中，我们无法保证在有序插入的前提下就能有序读出。最简单的一个理由就是，如果你把行记录插入块中，然后删除了该行，接下来插入的行会去填补块中的空余部分，这就无法保证有序了。具体见老师构造的例子如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t
  2  (a int,
  3  b varchar2(4000) default rpad('*',4000,'*'),
  4  c varchar2(3000) default rpad('*',3000,'*')
  5  );
表已创建。
SQL> insert into t (a) values (1);
已创建 1 行。
SQL> insert into t (a) values (2);
已创建 1 行。
SQL> insert into t (a) values (3);
已创建 1 行。
SQL> select A from t;
   A
-----
   1
   2
   3
SQL> delete from t where a=2;
已删除 1 行
SQL> insert into t (a) values (4);
已创建 1 行。
SQL> commit;
提交完成。
SQL> select A from t;
   A
-----
   1
   4
   3
```

脚本 4-11 测试表记录顺序插入却难以否保证顺序读出

收获，不止 Oracle

因为 BLOCK 大小默认是 8KB，我特意用 `rpada('*',4000,'*')`, `rpada('*',3000,'*')`来填充 B、C 字段，这样可以保证一个块只插入一条数据，方便做试验分析跟踪。老师本意就按 T 表 A 字段的序列顺序依次插入，然后很希望查询可以依据 A 字段的顺序展现 T 表记录，现在请大家仔细观察看看，有什么发现吗？”老师问。

“老师，如果有序应该展现 1，3，4，结果却是 1，4，3，说明无序了。”敬昱第一个回答。

“难道老师不是顺序插入的吗，为什么无序地展现呢？”老师继续问。

“因为第 2 条记录被删除了，第 4 条记录去填充第 2 条记录所在的块，所以无序了。”晶晶迅速接上老师的提问。

“回答得非常好，所以我们在查询数据时，如果想有序地展现，就必须使用 `order by`，否则根本不能保证顺序展现，而 `order by` 操作是开销很大的操作，具体开销可以看下面试验：

```
SQL> set linesize 1000
SQL> set autotrace traceonly
SQL> select A from t;
执行计划
-----
Plan hash value: 1601196873

-----
| Id | Operation          | Name | Rows | Bytes | Cost (%CPU) | Time      |
-----
| 0 | SELECT STATEMENT    |      | 3 | 39 | 3 (0)| 00:00:01 |
| 1 | TABLE ACCESS FULL | T    | 3 | 39 | 3 (0)| 00:00:01 |
-----

Note
-----
- dynamic sampling used for this statement
统计信息
-----
0 recursive calls
0 db block gets
7 consistent gets
0 physical reads
0 redo size
452 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
3 rows processed
```

```
SQL> select A from t order by A;
```

执行计划

Plan hash value: 961378228

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		3	39	4 (25)	00:00:01
1	SORT ORDER BY		3	39	4 (25)	00:00:01
2	TABLE ACCESS FULL	T	3	39	3 (0)	00:00:01

Note

- dynamic sampling used for this statement

统计信息

```

0 recursive calls
0 db block gets
7 consistent gets
0 physical reads
0 redo size
452 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
1 sorts (memory)
0 sorts (disk)
3 rows processed
```

脚本 4-12 比较有无 order by 语句在执行计划、开销的差异

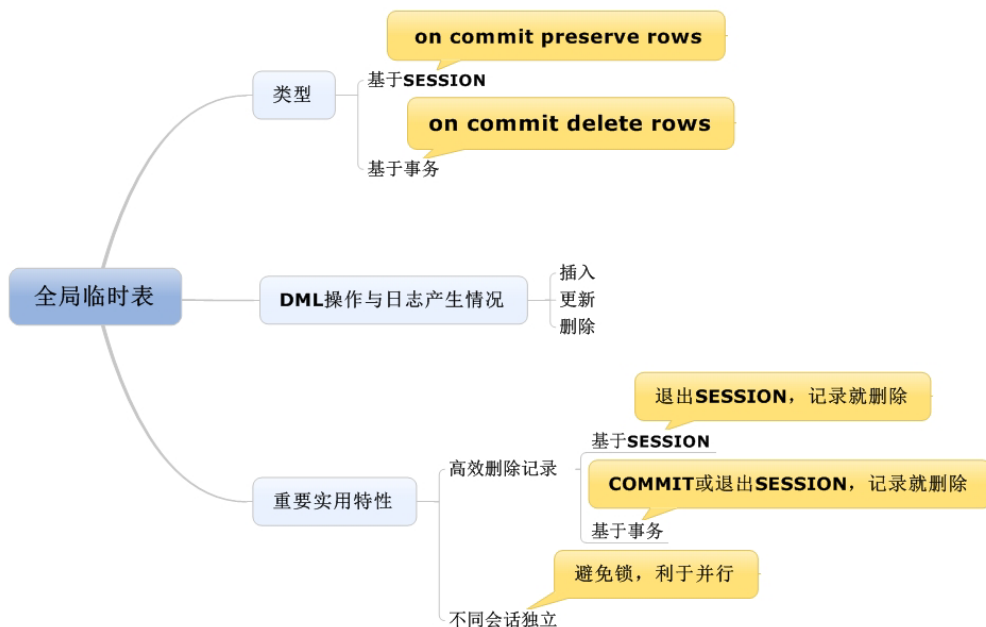
可以观察到，有排序的操作的统计信息模块有一个 1 sorts (memory)，表示发生了排序，执行计划中也有 SORT ORDER BY 的关键字，不过最重要的是，没排序的操作代价为 3，有排序的操作代价为 4，性能上是有差异的，在大数量时将会非常明显。

关于 order by 避免排序的方法有两种思路。第一种思路是在 order by 的排序列建索引，为什么可以消除排序呢？这就当成一个悬念，在后续索引的章节中给大家揭秘。第二种方法就是，将普通表改造为有序散列聚簇表，这样可以保证顺序插入，order by 展现时无须再有排序动作。

老师将在随后的章节中介绍有序散列聚簇表，这也是老师介绍的非普通堆组织表的最后一个部分，敬请期待。”

4.2.3 奇特的全局临时表

全局临时表的相关内容汇总如图 4-3 所示。



“从数据安全性来看，对表记录的操作写日志是不可避免的，否则备份恢复就无从谈起了，只是现实中我们真的有一部分应用对表的某些操作是不需要恢复的，比如运算过程中临时处理的中间结果集，这时我们就可以考虑用全局临时表来实现。

4.2.3.1 分析全局临时表的类型

全局临时表分为两种类型，一种是基于会话的全局临时表（**commit preserve rows**），一种是基于事务的全局临时表（**on commit delete rows**），具体建表语法见下面例子：

```
SQL> drop table t_tmp_session purge;
表已删除。
SQL> drop table t_tmp_transaction purge ;
表已删除。
SQL> create global temporary table T_TMP_session on commit preserve rows as select * from dba_objects
where 1=2;
表已创建。
SQL> select table_name,temporary,duration from user_tables where table_name='T_TMP_SESSION';
```

```

TABLE_NAME          T DURATION
-----
T_TMP_SESSION       Y SYS$SESSION
SQL> create global temporary table t_tmp_transaction on commit delete rows as select * from dba_objects where
1=2;
表已创建。
SQL> select table_name, temporary, DURATION from user_tables where table_name='T_TMP_TRANSACTION';
TABLE_NAME          T    DURATION
-----
T_TMP_TRANSACTION   Y    SYS$TRANSACTION

```

脚本 4-13 建基于事务和 SESSION 的全局临时表

上述命令完成了基于会话的全局临时表 T_TMP_SESSION 和基于事务的全局临时表 T_TMP_TRANSACTION。接下来大家肯定想知道 DML 操作针对全局临时表产生的日志和针对普通表有什么不同，是不是一定差别很大啊？”老师问。

“是！”大家都很肯定地回答，小莲有一种强烈的预感，全局临时表的 DML 操作肯定不会产生日志，她的预感正确吗？

4.2.3.2 观察各类 DML 的 REDO 量

“大家还记得老师在前面试验中构造的视图 v_redo_size 吧，下面的试验我们将用这个视图来分析日志产生的情况，这次试验是在建好上述两张全局临时表的基础上继续往下操作的，请大家仔细观察：

```

SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           65692
SQL> insert into t_tmp_transaction select * from dba_objects;
已创建 55752 行。
SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           369712
--注意：369712-65692=304020
SQL> insert into t_tmp_session select * from dba_objects;
已创建 55752 行。
SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           673624

```

收获，不止 Oracle

--注意：673624-369712=**303912**

SQL> update t_tmp_transaction set object_id=rownum;
已更新 55752 行。

SQL> select * from v_redo_size;

NAME	VALUE
redo size	3632516

--注意：更新 3632516-673624=**2958892**

SQL> update t_tmp_session set object_id=rownum;
已更新 55752 行。

SQL> select * from v_redo_size;

NAME	VALUE
redo size	8683824

--注意：8683824-3632516=**5051308**

SQL> delete from t_tmp_session;
已删除 55752 行。

SQL> select * from v_redo_size;

NAME	VALUE
redo size	25225220

--注意：删除会话全局临时表产生日志 25225220-8683824=**16541396**

SQL> delete from t_tmp_transaction;
已删除 55752 行。

SQL> select * from v_redo_size;

NAME	VALUE
redo size	41765456

---注意：删除事务全局临时表产生的日志 41765456-25225220=**16540236**

脚本 4-14 分别观察两种全局临时表针对各类 DML 语句产生的 redo 量

试验做完了，同学们是否看得有些眼花缭乱啊，大家认真观察一下，找找有什么发现。”老师问。

“老师，全局临时表没用啊，无论插入，修改还是删除，都还是要写日志啊。”小莲认真看了一会儿结果，有些大惑不解。

“老师，是不是全局临时表比普通表产生的日志要少呢，否则这还有什么性能上的优势啊？”晶晶也有些困惑。

“估计小莲同学是认为临时表不会写日志，所以发现会写日志后很失望。晶晶同学希望老师对相同表结构和表数据量的普通表进行操作，来比较产生日志的多少，那老师就先做试验如下：

```
SQL> drop table t purge;
```

表已删除。

```
SQL> create table t as select * from dba_objects where 1=2;
```

表已创建。

```
SQL> select * from v_redo_size;
```

NAME	VALUE
redo size	32680

```
SQL> insert into t select * from dba_objects;
```

已创建 55752 行。

```
SQL> select * from v_redo_size;
```

NAME	VALUE
redo size	6344836

--注意：插入普通表产生的日志是 6344836-32680=**6312156**

```
SQL> update t set object_id=rownum ;
```

已更新 55752 行。

```
SQL> select * from v_redo_size;
```

NAME	VALUE
redo size	14036720

--注意：更新普通表产生的日志是 14036720-6344836=**7691884**

```
SQL> delete from t;
```

已删除 55752 行。

```
SQL> select * from v_redo_size;
```

NAME	VALUE
redo size	34175980

--注意：删除普通表产生的日志为 34175980-14036720=**32772308**

脚本 4-15 全局临时表和普通表产生日志情况的比较

通过简单的比较我们即可得出结论：无论插入更新还是删除，操作普通表产生的日志都比全局临时表要多。”

看来自己的预感错了，小莲心想，DML 操作针对全局临时表来说，只是产生的日志要少得多，而不是不会产生。

4.2.3.3 全局临时表两大重要特性

1. 高效删除记录

“其实在我看来，全局临时表最重要的特点有两个。一是高效删除记录，基于事务的全局临时表 COMMIT 或者 SESSION 连接退出后，临时表记录自动删除；基于会话的全局临时表则是 SESSION 连接退出后，临时表记录自动删除，都无须我们动手去操作。二是针对不同会话数据独

收获，不止 Oracle

立，不同的 SESSION 访问全局临时表，看到的结果不同。

这两点既是全局临时表最重要的特点，也是最有用的特点，在工作中某些特定的场合，发挥出了巨大的作用。现在我们先从全局临时表的高效删除开始学习。

首先我们看看基于事务的全局临时表 COMMIT 后，记录是否已经删除了，并查看产生日志的情况，如下：

```
SQL> select count(*) from t_tmp_transaction;
COUNT(*)
-----
      0
SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           0
SQL> insert into t_tmp_transaction select * from dba_objects;
已创建 55721 行。
SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           303976
SQL> commit;
提交完成。
SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           304116
SQL> select count(*) from t_tmp_transaction;
COUNT(*)
-----
      0
```

脚本 4-16 基于事务的全局临时表的高效删除

这里大家有什么发现啊？”老师问。

“基于事务的全局临时表插入了 55721 行，COMMIT 后，再查这个表时记录就没了。”小莲第一个回答。

“还有，用 COMMIT 方式删除全局临时表记录所产生的日志量才 304116-303976=140，比起之前的直接用 delete 方式操作产生的日志量 16541396，几乎可以忽略不计了。”晶晶马上补充了小莲的回答。

“两位同学都回答得非常好，140 这个日志量其实是 COMMIT 动作本身产生的，所以我们基本可以理解为全局临时表的 COMMIT 或者退出 SESSION 的方式不会产生日志，这是 Oracle 的全

局临时表的一种特性。

接下来我们再看看基于 SESSION 的全局临时表的运行情况，如下：

```
SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           0
SQL> insert into t_tmp_session select * from dba_objects;
已创建 55721 行。
SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           304036
SQL> commit;
提交完成。
SQL> select count(*) from t_tmp_session;
COUNT(*)
-----
55721
SQL> select * from v_redo_size;
NAME                                VALUE
-----
redo size                           304176
```

脚本 4-17 基于 SESSION 的全局临时表 COMMIT 并不清空记录

同学们，现在有什么发现吗？”老师停下来问。

“老师，基于会话的全局临时表在 COMMIT 后表的记录依然存在，和基于事务的临时表真是不一样。另外我验证了刚才您说的 140 是 COMMIT 动作本身产生的 REDO，因为现在 304176-304036 也是等于 140。”晶晶回答得非常快，简直是脱口而出。

“正确，那我们再观察一下，看看退出 SESSION，再重新连接后，记录还在吗。

```
SQL> exit
C:\Users\ljb>sqlplus ljb/ljb
SQL> select count(*) from t_tmp_session;
COUNT(*)
-----
0
```

脚本 4-18 基于事务的全局临时表退出后再登入，观察记录情况

不用说大家也看得很清楚了，记录真的没有了，现在我再考考你们，大家现在都已经知道了

基于事务和基于会话（SESSION）的全局临时表的差别了，那什么时候用基于事务的，什么时候用基于 SESSION 的呢？”老师问。

“我知道，如果全局临时表在程序的一次调用执行过程中需要多次清空记录再插入记录，就要考虑用基于事务的，这时 COMMIT 可以把结果快速清理了，否则用 delete 效率低下。如果不存在这种情况，就用基于 SESSION 的，更简单，连 COMMIT 的动作都省了。”台下同学思考了好长一段时间，还是小莲先想明白了。

“说得非常好，一般来说，基于 SESSION 的全局临时表的应用会更多一些，少数比较复杂的应用，涉及一次调用中需要记录清空再插入等复杂动作时，才考虑用基于事务的全局临时表。

我在实际工作中经常使用到全局临时表，并发挥出了巨大的作用。在后续的课程中老师还会分享与全局临时表有关的经典案例，接下来我们来看看全局临时表的第二个重要特点，针对不同会话独立的有用特性。”

2. 不同会话独立

“下面我讨论全局临时表时，暂且就只讨论基于 SESSION 的临时表而不讨论基于事务的全局临时表了，因为它们基本上没什么大的差异。

首先我们先进入一个 SID=975 的 SESSION，完成全局临时表的插入，查看当前有记录，具体如下：

```
C:\Users\ljb>sqlplus ljb/ljb
SQL> select * from v$mystat where rownum=1;
      SID STATISTIC#      VALUE
-----
      975          0          1
SQL> select * from t_tmp_session;
未选定行
SQL> insert into t_tmp_session select * from dba_objects;
已创建 55721 行。
SQL> commit;
提交完成。
SQL> select count(*) from t_tmp_session;
COUNT(*)
-----
55721
```

脚本 4-19 基于全局临时表的会话独立性之观察第 1 个 SESSION

接下来再登录一个 SID=973 的新连接，发现同样是这个 t_tmp_session 表，记录为 0，对该表

插入一条，在当前 SESSION 中我们查询到该表就是一条记录。

```
C:\Users\ljb>sqlplus ljb/ljb
SQL> select * from v$mystat where rownum=1;
   SID   STATISTIC#    VALUE
-----
   973         0         1
SQL> select count(*) from t_tmp_session;
COUNT(*)
-----
        0
SQL> insert into t_tmp_session select * from dba_objects where rownum=1;
已创建 1 行。
SQL> commit;
提交完成。
SQL> select count(*) from t_tmp_session;
COUNT(*)
-----
        1
```

脚本 4-20 基于全局临时表的会话独立性之观察第 2 个 SESSION

假如这时我们回到 SID=975 的 SESSION 再去查看 t_tmp_session 表，我们会发现那个 SESSION 查询的依然是 55721 条。

这是一个神奇的特性，结合高效删除和灵活应用这两个特性，将会给相关工作带来巨大的帮助。大家将会在后续的课程中感受到。”

4.2.4 神通广大的分区表

“同学们，说完了普通表的不足以及全局临时表的神奇之处后，我们要进入本课程的重头戏环节，学习在表设计中极其重要一个技术：分区表设计（如图 4-4 所示）。

在如今数据量日益增长的海量数据库时代，分区表技术显得尤为重要，甚至可以说使用得当与否将决定到系统的生死。

分区表的特性总体是比较复杂的，过于详细的学习不符合老师的风格，所以我的课程不会面面俱到，大家回顾一下前面关于普通堆表不足的描述，其中表记录太大检索慢和 delete 删除有瑕疵这两个缺点正好可以被分区表的分区消除和可以高效清理分区数据这两大特点给弥补了。

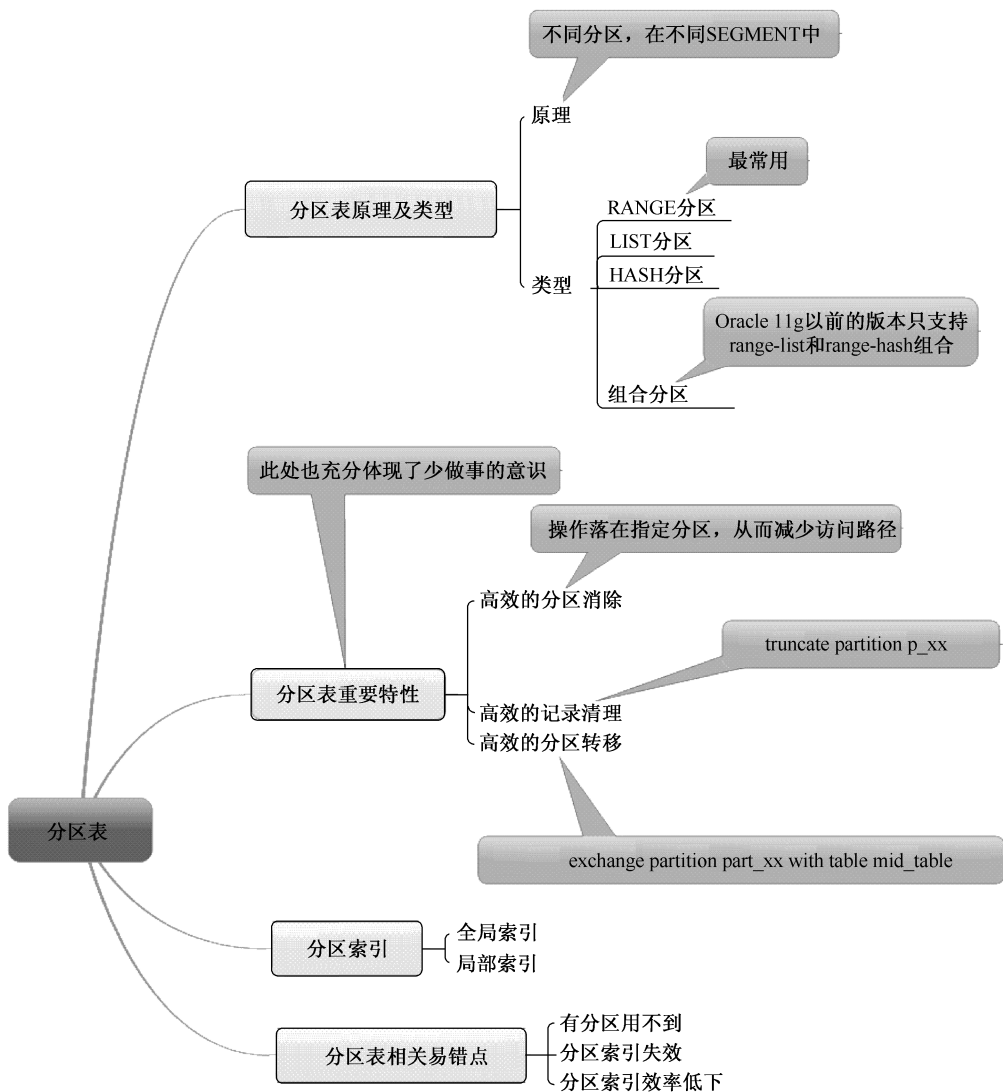


图 4-4 分区表

什么叫分区消除，最通俗的比喻就是，你对某表按月份建了范围分区，从 1 月到 12 月共 12 个分区，你查询当前 12 月的记录，就不会去访问另外 11 个区，少做事了，这就是分区消除。

那高效分区清理呢，就是如果你想删除某分区的数据，如果直接 `delete`，速度很慢，而且高水平位也不会释放，查询的块依然很多，前面已经见识过了，这时可以直接 `truncate` 这个分区，速度非常快。

虽然听起来轻描淡写的似乎很简单，但是我时常发现不少技术人员项目应用中都不善于利用

分区表的这些特性，最终遭遇性能瓶颈，甚是可惜。

此外这里老师不厌其烦地再强调一点，大家应该都知道此次系列课程的风格，我是不会专程去讲述语法的，但是同学们可以在试验脚本中自行体会到。我也不会讲述所有知识，但是会给大家指明学习路线。因为课堂上没有那么多的时间，此外大家别忘记了在这个 Google 的时代，语法和知识点都不是问题，搜不到的是体系，是重点，是思想。”

小莲感触颇深地点了点头，要是依次讲述语法，罗列所有知识，她早就不来了。她看重的是老师授课中的重点与体系，应用及思想。

4.2.4.1 分区表类型及原理

“好了，现在老师开始描述分区表了，首先探讨的是分区表的类型及原理。分区表的类型有范围分区、列表分区、HASH 分区及组合分区 4 种，其中范围分区应用最为广泛，需要重点学习和掌握，而列表分区次之，在某些场合下也可以考虑使用组合分区（在 Oracle 11g 以前，组合分区的组合方式比较有限），相对而言 HASH 分区在应用中适用的场景并不广泛，使用的频率比较低，故在本次课程中暂且不予以细致的描述，有兴趣的同学可以自行通过文档研究学习。

现在老师通过一组试验来给大家展现这些分区表的使用，请大家认真观察脚本，先了解这些分区表的建立方法。

1. 范围分区

首先是看一组范围分区的例子，记住，范围分区最常见的是按时间列进行分区，如下：

```
SQL> drop table range_part_tab purge;
```

表已删除。

```
SQL> --注意，此分区为范围分区
```

```
SQL> create table range_part_tab (id number,deal_date date,area_code number,contents varchar2(4000))
```

```
2      partition by range (deal_date)
3      (
4          partition p1 values less than (TO_DATE('2012-02-01', 'YYYY-MM-DD')),
5          partition p2 values less than (TO_DATE('2012-03-01', 'YYYY-MM-DD')),
6          partition p3 values less than (TO_DATE('2012-04-01', 'YYYY-MM-DD')),
7          partition p4 values less than (TO_DATE('2012-05-01', 'YYYY-MM-DD')),
8          partition p5 values less than (TO_DATE('2012-06-01', 'YYYY-MM-DD')),
9          partition p6 values less than (TO_DATE('2012-07-01', 'YYYY-MM-DD')),
10         partition p7 values less than (TO_DATE('2012-08-01', 'YYYY-MM-DD')),
11         partition p8 values less than (TO_DATE('2012-09-01', 'YYYY-MM-DD')),
12         partition p9 values less than (TO_DATE('2012-10-01', 'YYYY-MM-DD')),
13         partition p10 values less than (TO_DATE('2012-11-01', 'YYYY-MM-DD')),
14         partition p11 values less than (TO_DATE('2012-12-01', 'YYYY-MM-DD')),
```

```
15      partition p12 values less than (TO_DATE('2013-01-01', 'YYYY-MM-DD')),
16      partition p_max values less than (maxvalue)
17  )
18  ;
```

表已创建。

SQL>

SQL> --以下是插入 2012 年一整年日期随机数和表示福建地区号含义（591 到 599）的随机数记录，共有 10 万条，如下：

SQL> insert into range_part_tab (id,deal_date,area_code,contents)

```
2      select rownum,
3             to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)), 'J'),
4             ceil(dbms_random.value(590,599)),
5             rpad('*',400,'*')
6      from dual
7      connect by rownum <= 100000;
```

已创建 100000 行。

SQL> commit;

提交完成。

脚本 4-21 范围分区示例

以上操作完成了范围分区的分区表建表的示例，并且构造出 10 万条记录插入到分区表中，注意如下 5 点：

- ① 范围分区的关键字为 **partition by range**，即这三个关键字表示该分区为范围分区。
- ② **values less than** 是范围分区特定的语法，用于指明具体的范围，比如 **partition p2 values less than (TO_DATE('2012-03-01', 'YYYY-MM-DD'))**，表示小于 3 月份的记录。
- ③ **partition p1** 到 **partition p_max** 表示总共建立了 13 个分区。
- ④ 最后还要注意 **partition p_max values less than (maxvalue)** 的部分，表示超出这些范围的记录全部落在这个分区中，免得出错。
- ⑤ 分区表的分区可分别指定在不同的表空间里，如果不写即为都在同一默认表空间里。

请同学们现在认真观察一分钟，回去后再抽空自行运行老师提供的这些脚本。”老师停下来让同学们仔细观察。

2. 列表分区

“接下来我们看一组列表分区的例子，表名虽然变为 **list_part_tab**，但是字段和插入记录的情

况却是相同的，明显的区别在于建分区的字段不再是时间列，而是表示地区号的列，一般来说，列表分区最常见的分区列就是以地区列作为分区，如下：

```
SQL> drop table list_part_tab purge;
表已删除。
SQL> --注意，此分区为列表分区
SQL> create table list_part_tab (id number,deal_date date,area_code number,contents varchar2(4000))
2      partition by list (area_code)
3      (
4      partition p_591 values (591),
5      partition p_592 values (592),
6      partition p_593 values (593),
7      partition p_594 values (594),
8      partition p_595 values (595),
9      partition p_596 values (596),
10     partition p_597 values (597),
11     partition p_598 values (598),
12     partition p_599 values (599),
13     partition p_other values (DEFAULT)
14     )
15     ;
```

表已创建。

SQL>

SQL>

SQL> --以下是插入 2012 年一整年日期随机数和表示福建地区号含义（591 到 599）的随机数记录，共有 10 万条，如下：

```
SQL> insert into list_part_tab (id,deal_date,area_code,contents)
2      select rownum,
3      to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)), 'J'),
4      ceil(dbms_random.value(590,599)),
5      rpad('*',400,'*')
6      from dual
7      connect by rownum <= 100000;
```

已创建 100000 行。

SQL> commit;

提交完成。

脚本 4-22 列表分区示例

收获，不止 Oracle

以上操作完成了列表分区的分区表建表的示例，构造与之前相同的 10 万条记录插入到分区表中，同样需注意如下几点：

- ① 列表分区的关键字为 `partition by list`，即这三个关键字表示该分区为列表分区。
- ② 不同于之前范围分区的 `values less than`，列表分区仅需 `values` 即可确定范围，值得注意的是，`partition p_592 values (592)` 并不是说明取值只能写一个，也可写为多个，比如 `partition p_union values (592,593,594)`。
- ③ `partition p_591` 到 `partition p_other` 表示总共建立了 10 个分区。
- ④ 最后还要注意 `partition p_other values (default)` 的部分，表示不在刚才 591 到 599 范围的记录全部落在这个默认分区中，避免应用出错。
- ⑤ 分区表的分区可分别指定在不同的表空间里，如果不写即为都在同一默认表空间里。

请同学们和之前一样认真观察一分钟，体会一下列表分区的特别之处。”老师再次停下来让同学们仔细观察。

3. 散列分区

“接下来我们继续看看散列分区（HASH 分区）的例子，和之前一样，表名虽然变为 `hash_part_tab`，但是字段和插入记录的情况却都是相同的，这次老师建散列分区时所取的列故意是和之前范围分区相同的列，都是时间列，如下：

```
SQL> drop table hash_part_tab purge;
```

表已删除。

```
SQL> --注意，此分区 HASH 分区
```

```
SQL> create table hash_part_tab (id number,deal_date date,area_code number,contents varchar2(4000))
```

```
2      partition by hash (deal_date)
```

```
3      PARTITIONS 12
```

```
4      ;
```

表已创建。

```
SQL> --以下是插入 2012 年一整年日期随机数和表示福建地区号含义（591 到 599）的随机数记录，共有 10 万条，如下：
```

```
SQL> insert into hash_part_tab(id,deal_date,area_code,contents)
```

```
2      select rownum,  
3          to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)), 'J'),  
4          ceil(dbms_random.value(590,599)),  
5          rpad('*',400,'*')  
6      from dual  
7      connect by rownum <= 100000;
```

已创建 100000 行。

```
SQL> commit;
提交完成。
```

脚本 4-23 散列分区示例

以上操完成了散列分区表的建立和 10 万条记录的插入。怎么样，是不是觉得这个散列分区好像比起之前的两种类型的分区，要简单得多啊。”老师笑着问。

同学们还没反应过来，都认真地看着屏幕。

“针对散列分区，我也说几个需要注意的重点，如下。

- ① 散列分区的关键字为 **partition by hash**，出现这三个关键字即表示当前分区为散列分区。
- ② 散列分区与之前两种分区的明显差别在于：没有指定分区名，而仅仅是指定了分区个数，如 **PARTITIONS 12**。
- ③ 散列分区的分区个数尽量设置为偶数个，比如本例是 12 个分区，如果是 11 个或者 13 个就不妥了，具体原因和 Oracle 内部架构有关，这里就不再细说。
- ④ 可以指定散列分区的分区表空间，比如增加如下一小段，**STORE IN (ts1,ts2,ts3,ts4,t5,t6,t7,t8,t9,t10,t11,t12)**表示分别存在在 12 个不同的表空间里，当然不写出表空间就是都在同一默认表空间里。

请同学们还是再仔细观察大屏幕一分钟，认真体会散列分区的特别之处。”

4. 组合分区

“接下来我们看看组合分区的例子，在 Oracle 11g 以前主要支持范围-列表和范围-散列这两种组合，在实际应用中最常用的组合是范围-列表（range-list）的组合，我们就先来看一组与之相关的组合分区的试验，如下：

```
SQL> drop table range_list_part_tab purge;
表已删除。
SQL> --注意，此分区为范围分区
SQL> create table range_list_part_tab (id number,deal_date date,area_code number,contents varchar2(4000))
2      partition by range (deal_date)
3      subpartition by list (area_code)
4      subpartition TEMPLATE
5      (subpartition p_591 values (591),
6          subpartition p_592 values (592),
7          subpartition p_593 values (593),
8          subpartition p_594 values (594),
9          subpartition p_595 values (595),
10         subpartition p_596 values (596),
11         subpartition p_597 values (597),
12         subpartition p_598 values (598),
```



```
13         subpartition p_599 values (599),
14         subpartition p_other values (DEFAULT))
15     (
16         partition p1 values less than (TO_DATE('2012-02-01', 'YYYY-MM-DD')),
17         partition p2 values less than (TO_DATE('2012-03-01', 'YYYY-MM-DD')),
18         partition p3 values less than (TO_DATE('2012-04-01', 'YYYY-MM-DD')),
19         partition p4 values less than (TO_DATE('2012-05-01', 'YYYY-MM-DD')),
20         partition p5 values less than (TO_DATE('2012-06-01', 'YYYY-MM-DD')),
21         partition p6 values less than (TO_DATE('2012-07-01', 'YYYY-MM-DD')),
22         partition p7 values less than (TO_DATE('2012-08-01', 'YYYY-MM-DD')),
23         partition p8 values less than (TO_DATE('2012-09-01', 'YYYY-MM-DD')),
24         partition p9 values less than (TO_DATE('2012-10-01', 'YYYY-MM-DD')),
25         partition p10 values less than (TO_DATE('2012-11-01', 'YYYY-MM-DD')),
26         partition p11 values less than (TO_DATE('2012-12-01', 'YYYY-MM-DD')),
27         partition p12 values less than (TO_DATE('2013-01-01', 'YYYY-MM-DD')),
28         partition p_max values less than (maxvalue)
29     )
30 ;
```

表已创建。

SQL> --以下是插入 2012 年一整年日期随机数和表示福建地区号含义（591 到 599）的随机数记录，共有 10 万条，如下：

```
SQL> insert into range_list_part_tab(id,deal_date,area_code,contents)
2     select rownum,
3         to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)), 'J'),
4         ceil(dbms_random.value(590,599)),
5         rpad('*',400, '*')
6     from dual
7     connect by rownum <= 100000;
```

已创建 100000 行。

SQL> commit;

提交完成。

脚本 4-24 组合分区示例

以上试验完成了组合分区表（范围-列表）的建立和 10 万条记录的插入。刚才大家看完貌似语法最简单的散列分区后，现在老师给大家展现的是看上去写法最复杂的组合分区，是不是有些头昏脑胀？

不过大家仔细看看就会发现，其实主要也就增加了 **subpartition by list (area_code)** 这个模块，其他部分和原先的范围分区没啥差异。这里大家要注意如下几个重点。

- ① 组合分区是由主分区和从分区组成的，比如范围-列表分区，就表示主分区是范围分区，而从分区是列表分区，从分区的关键字为 **subpartition**，比如本例中的 **subpartition by list**

(area_code)。

- ② 为了避免在每个主分区中都写相同的从分区，可以考虑用模版方式，比如本例中的 subpartition TEMPLATE 关键字。
- ③ 只要涉及子分区模块，都需要有 subpartition 关键字。
- ④ 关于表空间和之前的没有差别，依然是可以指定，也可以不指定。

Oracle 11g 以前，除了上述范围-列表这个最常见的组合分区外，还有范围-HASH 组合。在 Oracle 11g 后，还提供了 RANGE-RANGE、LIST-RANGE、LIST-HASH 和 LIST-LIST 这 4 种组合，本次课程我就主要说明范围-列表分区，其他的适用的场合相对比较少，使用频率较低，我就不再一一举例了。

同学们可能观察和学习这 4 个试验脚本已经有些头昏脑胀了，大家休息一下，十分钟后我们继续。接下来的课程将在前面这些试验的基础上继续，探讨分区表的原理以及分区表有哪些特性，体会分区表设计到底能给我们带来什么好处。”

5. 分区原理

“大家好，现在我们开始探讨分区表的原理是什么，实践出真知，我们再来看一组试验，最终结论还是从试验中得出。

前面建了 4 种不同类型的分区表并依次插入了相同的 10 万条记录，现在再建一张普通表，也插入相同的 10 万条记录，如下：

```
SQL> drop table norm_tab purge;
表已删除。
SQL> create table norm_tab (id number,deal_date date,area_code number,contents varchar2(4000));
表已创建。
SQL> insert into norm_tab(id,deal_date,area_code,contents)
2   select rownum,
3         to_date( to_char(sysdate-365,'J')+TRUNC(DBMS_RANDOM.VALUE(0,365)), 'J'),
4         ceil(dbms_random.value(590,599)),
5         rpad('*',400,'*')
6   from dual
7   connect by rownum <= 100000;
已创建 100000 行。
SQL> commit;
提交完成。
```

脚本 4-25 分区原理分析之普通表插入

接下来我们来观察普通表和分区表在段的分配上有何差异，此处仅以范围分区表来和普通表进行试验比较，如下：

```
SQL> SET LINESIZE 666
```

```
SQL> set pagesize 5000
SQL> column segment_name format a20
SQL> column partition_name format a20
SQL> column segment_type format a20
SQL> select segment_name,
2         partition_name,
3         segment_type,
4         bytes / 1024 / 1024 "字节数(M)",
5         tablespace_name
6     from user_segments
7    where segment_name IN('RANGE_PART_TAB','NORM_TAB');
```

SEGMENT_NAME	PARTITION_NAME	SEGMENT_TYPE	字节数(M)	TABLESPACE_NAME
NORM_TAB		TABLE	47	TBS_LJB
RANGE_PART_TAB	P1	TABLE PARTITION	6	TBS_LJB
RANGE_PART_TAB	P2	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P3	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P4	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P5	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P6	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P7	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P8	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P9	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P10	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P11	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P12	TABLE PARTITION	3	TBS_LJB
RANGE_PART_TAB	P_MAX	TABLE PARTITION	.0625	TBS_LJB

已选择 14 行。

脚本 4-26 分区原理分析之普通表与分区表在段分配上的差异

这里大家有什么发现呢？”老师问。

“老师，这个试验说了分区表会产生多个 SEGMENT，而且是建了几个分区就有几个 SEGMENT，而普通表仅有一个 SEGMENT。”林君最先回答。

“说得非常好！”老师满意地点点头。

“还有，您前面说的分区表可以指定不同的表空间，在不写明的情况下，都是指向默认表空间的，这一结论似乎可以在这里得以证明。”晶晶总是比较细心。

“没错，晶晶同学说得对，如果我的分区表建立之初，在每个分区上都指定不同的表空间名，大家在此处 TABLESPACE_NAME 列看到的就不会是清一色的 TBS_LJB 表空间了。

刚才林君同学的回答其实已经点明了分区表存在的意义。很简单的一个思想：化整为零，将

大对象切割成多个小对象，从而使得在指定的小对象中定位到数据成为一种可能，最终达到减少访问路径，尽量少做事就能解决问题的目的。”

同学们纷纷点头称是。

“刚才老师只是让大家查看了普通表和范围分区表的段的分配情况，现在我们再来看看散列分区的试验情况，请大家仔细观察，如下：

```
SQL> SET LINESIZE 666
SQL> set pagesize 5000
SQL> column segment_name format a20
SQL> column partition_name format a20
SQL> column segment_type format a20
SQL> select segment_name,
2      partition_name,
3      segment_type,
4      bytes / 1024 / 1024 "字节数(M)",
5      tablespace_name
6  from user_segments
7  where segment_name IN('HASH_PART_TAB');
```

SEGMENT_NAME	PARTITION_NAME	SEGMENT_TYPE	字节数(M)	TBSPACE_NAME
HASH_PART_TAB	SYS_P15733	TABLE PARTITION	4	TBS_LJB
HASH_PART_TAB	SYS_P15734	TABLE PARTITION	3	TBS_LJB
HASH_PART_TAB	SYS_P15735	TABLE PARTITION	2	TBS_LJB
HASH_PART_TAB	SYS_P15736	TABLE PARTITION	4	TBS_LJB
HASH_PART_TAB	SYS_P15737	TABLE PARTITION	5	TBS_LJB
HASH_PART_TAB	SYS_P15738	TABLE PARTITION	7	TBS_LJB
HASH_PART_TAB	SYS_P15739	TABLE PARTITION	6	TBS_LJB
HASH_PART_TAB	SYS_P15740	TABLE PARTITION	6	TBS_LJB
HASH_PART_TAB	SYS_P15741	TABLE PARTITION	4	TBS_LJB
HASH_PART_TAB	SYS_P15742	TABLE PARTITION	4	TBS_LJB
HASH_PART_TAB	SYS_P15743	TABLE PARTITION	4	TBS_LJB
HASH_PART_TAB	SYS_P15744	TABLE PARTITION	4	TBS_LJB

已选择 12 行。

脚本 4-27 观察 HASH 分区的段分配情况

大家注意了，此次的 HASH 分区和范围分区的分区列是同一字段，都是 deal_date 这个时间字段，现在请同学们认真看看屏幕的结果，说说这个 HASH 分区有什么特别之处。”老师问。

“老师，PARTITON_NAME 全是系统自带的，不是我们指定的。”曾祥说话总是大着嗓门。

“说得很对，你在建 HASH 分区时，有指定分区名吗？”老师反问。

“没有！”曾祥又回头看了一眼老师教材中的试验。

“所以，分区名当然是系统自己命名的了。”老师笑了。

大家也都跟着笑了。

“不过话说回来，老师目的就是要让大家发现这点，这里说明了 HASH 分区表的一个弱点，就是无法让指定的数据到指定的分区去，这对我们快速检索数据并不是很有利，因此 HASH 分区在实际的工作中应用得相对较少一些。

不过任何事情存在即合理，大家想想看，HASH 分区该应用在什么场合下呢？”老师问。

同学们一时没想明白，台下无人应答。

“其实 HASH 分区最大的好处在于，将数据根据一定 HASH 算法，均匀分布到不同的分区中去，避免查询数据时集中在某一个地方，从而避免热点块的竞争，改善 IO，此处老师的时间列建 HASH 分区一般是不妥当的，因为我们经常都是指定具体的时间来完成数据检索，或者是指定具体的时间来完成数据清理，这对 HASH 分区来说，就很不适合了，此外大家还要注意一点，HASH 可以精确匹配，无法范围扫描。

现实中我们可以针对某些本身就无法有效执行分区范围的列进行 HASH 分区，比如 ID 列之类的，在出现热点块竞争严重时，可考虑如此设计。

现在让我们再看看组合分区的情况，请看屏幕上一个试验，然后说说自己体会，如下：

```
SQL> select count(*)  
2    from user_segments  
3    where segment_name = 'RANGE_LIST_PART_TAB';  
COUNT(*)
```

130

脚本 4-28 观察组合分区的段分配的个数

同学们，有什么发现吗？”老师问。

“非组合分区表查询 user_segments 时才占用了数十个分区，而这一组合，居然有 130 个！”小莲看明白后，吃惊地喊出来。

“这里也说明了组合分区存在的意义，就是化整为零思想的升级版，将一个大对象切割得更细更小了。这对于一个超级大表来说，也是有一定的意义的。

当然，在表特别大时，将对象切割得更细还是有不少方法，比如原先是按月分区，改为按天分区后，分区数量自然也就更多了。现实中，在某些非常特别的应用场合下，也有过按小时分区的个别案例，不过一般比较少见。

大家记住了，分区表也是有额外开销的，如果分区数量过多，Oracle 就需要管理过多的段，在操作分区表时也容易引发 Oracle 内部大量的递归调用，此外本身的语法也有一定的复杂度。所

以一般来说，只有大表才建议建分区，记录数在 100 万以下的表，基本上不建议建分区。

本小节的学习是让大家通过试验初步掌握建立各类分区的方法，以及理解分区表的原理和存在的意义，下面我们将在上述试验的基础上继续深入，开始揭开分区表最实用的几个特性。”

4.2.4.2 分区表最实用的特性

1. 高效的分区消除

“分区表存在最大的意义就在于，可以有效地做到分区消除，比如你对地区号做了分区，查询福州就只会在福州的分区中查找数据，而不会到厦门、漳州、泉州等其他分区中查找，这就是分区消除，消除了福州以外的所有其他分区。

原理很简单，就是因为分区表其实是将一个大对象分成了多个小对象，具体试验在前面的分区原理中已经介绍过了。

现在我们来看一组试验，看看分区消除是怎样给我们带来性能上的大幅提升的，这里参与试验的两张表分别是分区表 `range_part_tab` 和普通表 `norm_tab`，在前面的试验中已经建过这两张表，并分别插入过 10 万条记录，大家应该还记得。

首先是跟踪按时间范围分区的 `range_part_tab` 的 SQL 执行性能情况，如下：

```
SQL> set linesize 1000
```

```
SQL> set autotrace traceonly
```

```
SQL> set timing on
```

```
SQL> select *
```

```
2   from range_part_tab
3   where deal_date >= TO_DATE('2012-09-04', 'YYYY-MM-DD')
4     and deal_date <= TO_DATE('2012-09-07', 'YYYY-MM-DD');
```

已选择 1104 行。

执行计划

```
-----
Plan hash value: 16125146
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	t	Pstop
0	SELECT STATEMENT		1165	2302K	111 (1)	00:00:02			
1	PARTITION RANGE SINGLE		1165	2302K	111 (1)	00:00:02	9		9
* 2	TABLE ACCESS FULL	RANGE_PART_TAB	1165	2302K	111 (1)	00:00:02	9		9

Predicate Information (identified by operation id):

```
2 - filter("DEAL_DATE">=TO_DATE(' 2012-09-04 00:00:00', 'syyy-MM-dd hh24:mi:ss') AND
          "DEAL_DATE"<=TO_DATE(' 2012-09-07 00:00:00', 'syyy-MM-dd hh24:mi:ss'))
```

Note

收获，不止 Oracle

- dynamic sampling used for this statement
统计信息

0 recursive calls
0 db block gets
566 consistent gets
0 physical reads
0 redo size
26236 bytes sent via SQL*Net to client
1203 bytes received via SQL*Net from client
75 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1104 rows processed

脚本 4-29 观察范围分区表的分区消除带来的性能优势

接下来执行同样的语句，只是表名是普通表 `norm_tab`，继续性能测试如下：

```
SQL> select *  
2   from norm_tab  
3   where deal_date >= TO_DATE('2012-09-04', 'YYYY-MM-DD')  
4   and deal_date <= TO_DATE('2012-09-07', 'YYYY-MM-DD');
```

已选择 1011 行。

执行计划

Plan hash value: 278673677

	Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
	0	SELECT STATEMENT		1126	2225K	1298 (1)	00:00:16
*	1	TABLE ACCESS FULL	NORM_TAB	1126	2225K	1298 (1)	00:00:16

Predicate Information (identified by operation id):

1 - filter("DEAL_DATE">=TO_DATE(' 2012-09-04 00:00:00', 'syyyy-mm-dd
hh24:mi:ss')) AND "DEAL_DATE"<=TO_DATE(' 2012-09-07 00:00:00',
'syyyy-mm-dd hh24:mi:ss'))

Note

- dynamic sampling used for this statement
统计信息

```

-----
0 recursive calls
0 db block gets
5990 consistent gets
5457 physical reads
0 redo size
24261 bytes sent via SQL*Net to client
1137 bytes received via SQL*Net from client
69 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1011 rows processed

```

脚本 4-30 比较相同语句，普通表无法用到 DEAL_DATE 条件进行分区消除的情况

怎么样，有什么发现？”老师问。

“老师，同样的语句查询相同记录的表，分区表的查询代价仅为 111，逻辑读仅为 566，而普通表代价却为 1298，逻辑读为 5990，性能方面有着天壤之别。

差别如此之大，应该是和分区表查询只遍历了 13 个分区中的一个有关。在分区表查询的执行计划中看到 p_start 和 p_stop 都标记上 9，表示只遍历了第 9 个分区。这样避开了对其余 12 个分区的查询，就是您所说的分区消除吧。”晶晶观察能力和总结能力都很强。

“说得太好了，老师都没什么可补充的了。”老师竖起大拇指表扬了晶晶同学。

“接下来继续探讨分区消除，还记得老师之前说过的组合分区吧，大家来观察一下组合分区试验情况。

首先是针对分区表的查询，这次语句的表从 range_part_tab 更改为 range_list_part_tab，从范围分区表变为范围-列表的组合分区表，而且 SQL 语句也略有变化，增加了 area_code 查询条件，如下：

```

SQL> set autotrace traceonly
SQL> set linesize 1000
SQL> select *
2   from range_list_part_tab
3  where deal_date >= TO_DATE('2012-09-04', 'YYYY-MM-DD')
4     and deal_date <= TO_DATE('2012-09-07', 'YYYY-MM-DD')
5     and area_code=591;

```

已选择 101 行。

执行计划

```

-----
Plan hash value: 406789865

```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
----	-----------	------	------	-------	-------------	------	--------	-------

	0	SELECT STATEMENT		267		527K		114	(0)	00:00:02			
	1	PARTITION RANGE SINGLE		267		527K		114	(0)	00:00:02		9	9
	2	PARTITION LIST SINGLE		267		527K		114	(0)	00:00:02		KEY	KEY
*	3	TABLE ACCESS FULL	RANGE_LIST_PART_TAB		267		527K		114	(0)	00:00:02		81 81

Predicate Information (identified by operation id):

3 - filter("DEAL_DATE">=TO_DATE(' 2012-09-04 00:00:00', 'yyyy-mm-dd hh24:mi:ss') AND
"DEAL_DATE"<=TO_DATE(' 2012-09-07 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))

Note

- dynamic sampling used for this statement

统计信息

0 recursive calls
0 db block gets
60 consistent gets
0 physical reads
0 redo size
2844 bytes sent via SQL*Net to client
466 bytes received via SQL*Net from client
8 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
101 rows processed

脚本 4-31 观察 LIST 分区表的分区条件的分区消除

接下来测试针对普通表的增加了 area_code 条件的相同 SQL 语句，性能测试如下：

SQL> select *
2 from norm_tab
3 where deal_date >= TO_DATE('2012-09-04', 'YYYY-MM-DD')
4 and deal_date <= TO_DATE('2012-09-07', 'YYYY-MM-DD')
5 and area_code=591;

已选择 107 行。

已用时间: 00: 00: 01.35

执行计划

Plan hash value: 278673677

	Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	
--	----	-----------	------	------	-------	-------------	------	--

```
| 0 | SELECT STATEMENT |          | 282 | 557K | 1298 (1) | 00:00:16 |
|* 1 | TABLE ACCESS FULL | NORM_TAB | 282 | 557K | 1298 (1) | 00:00:16 |
```

Predicate Information (identified by operation id):

```
1 - filter("AREA_CODE"=591 AND "DEAL_DATE">=TO_DATE(' 2012-09-04
      00:00:00', 'yyyy-mm-dd hh24:mi:ss') AND "DEAL_DATE"<=TO_DATE('
      2012-09-07 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

Note

- dynamic sampling used for this statement

统计信息

```
0 recursive calls
0 db block gets
5931 consistent gets
5673 physical reads
0 redo size
3057 bytes sent via SQL*Net to client
477 bytes received via SQL*Net from client
9 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
107 rows processed
```

脚本 4-32 比较相同语句，普通表无法用 area_code 条件进行分区消除的情况

这里大家应该看得很清楚，组合分区导致范围定位得更小了，只产生了 60 个逻辑读，比之前的 566 个逻辑读还要少。至于普通表的查询，因为始终是全表扫描，所以逻辑读依然和之前差不多，达到 5931 个。

不过大家认真看看就会发现，组合分区虽然只有 60 个逻辑读，比之前范围分区的 566 个要小得多，但是代价 COST 却差不多，甚至 114 还略大于之前的 111，知道这是怎么回事吗？”老师问。

“是因为分区数过多，调用有开销吧，我记得之前您的试验结果是范围-列表分区表的 SEGMENT 段总数有 130 个，是范围分区的 10 倍大。”小莲的回答略微有些迟疑。

“没错，正是这个原因，由于在我们的试验中，表总记录不过 10 万，全表扫描开销都不是太大，这时 Oracle 内部调用的开销影响就相对较大。如果表是一张超级大表，比如有上亿，那这些开销相比而言就可以忽略不计了，这就是老师之前强调过的，分区表应用在大表更合适，至少要大于 100 万条记录的表方可考虑。”老师肯定了小莲的回答并做了补充。

收获，不止 Oracle

2. 强大的分区操作

(1) 分区 truncate 好快捷

“大家还记得在描述普通堆表时老师有说过，delete 无法释放空间，而 truncate 却有效地释放了空间。但可惜的是，针对普通表而言，truncate 往往不能轻易使用，因为 delete 往往是针对某些条件的局部记录删除，而 truncate 显然不能带上条件，无法做到局部删除。

这时分区表就发挥作用了，Oracle 可以实现只 truncate 某个分区，这就等同于实现了局部删除，我们来看一组简单的试验，如下：

```
SQL> delete from norm_tab
      where deal_date>=TO_DATE('2012-09-01', 'YYYY-MM-DD') and deal_date <= TO_DATE('2012-09-30', 'YYYY-MM-DD');
已删除 8111 行。
--为了后续章节试验的方便，本处暂且将删除的记录回退。
SQL> rollback;
回退已完成。

SQL> alter table range_part_tab truncate partition p9;
表被截断。
SQL> select count(*) from range_part_tab
      where deal_date>=TO_DATE('2012-09-01', 'YYYY-MM-DD') and deal_date <= TO_DATE('2012-09-30',
'YYYY-MM-DD');
COUNT(*)
-----
0
```

脚本 4-33 分区清除的方便例子

这个例子非常简单，说明了针对普通表我们只能 delete，而针对分区表，需求等同于要删除 p9 的分区，然后直接 alter table range_part_tab truncate partition p9 就完成了。关于 truncate 可以释放表空间之前已经证明了，现在无须证明了。

分区清理的方法，在有大量历史数据需要清理时候，发挥着极其重要的作用。很多历史表、日志表都被设计为分区表，正是由于这个特性使得清理数据极其方便迅速，而且能有效释放空间。”

“老师，如果历史数据不想删除，而想备份出去，那和普通表的操作相比，分区表是否就没什么优势了？”小莲忽然想到这点，随即脱口而出。

“不错，很有想法，老师正准备讲这方面的内容。”老师笑着说。

(2) 分区数据转移很神奇

“关于分区表的历史记录的处理，其实是可以分成删除和转移两部分的，关于转移备份的方案，Oracle 提供了一个非常棒的工具，就是分区交换，可以实现普通表和分区表的某个分区之间

数据的相互交换，他们之间的交换非常快，基本上在瞬间就可以完成，实际上只是 Oracle 在内部数据字典做的一些小改动而已。命令很简单，类似：`alter table 分区表 exchange partition 分区名 with table 中间表`。其中的关键字为 `exchange` 和 `with table` 等。

不过要注意的一点是，这两张表的字段必须是完全一样的，下面我们共同研究一组试验，来看看分区交换的神奇之处，如下：

```
SQL> drop table mid_table purge;
表已删除。
SQL> create table mid_table (id number deal_date date,area_code number,contents varchar2(4000));
表已创建。
SQL> select count(*) from mid_table ;
COUNT(*)
-----
0
SQL> select count(*) from range_part_tab partition(p8);
COUNT(*)
-----
8628
--当然，除了上述用 partition(p8)的指定分区名查询外，也可以采用分区条件代入查询：
SQL> select count(*) from range_part_tab
      where deal_date>=TO_DATE('2012-08-01', 'YYYY-MM-DD') and deal_date <= TO_DATE('2012-08-31',
'YYYY-MM-DD');
COUNT(*)
-----
8628
--以下命令就是经典的分区交换：
SQL> alter table range_part_tab exchange partition p8 with table mid_table;
表已更改。
--查询发现分区 8 数据不见了。
SQL> select count(*) from range_part_tab partition(p8);
COUNT(*)
-----
0
--而普通表记录由刚才的 0 条变为 8628 条了，果然实现了交换。
SQL> select count(*) from mid_table ;
COUNT(*)
-----
8628
```

脚本 4-34 分区交换的神奇例子

怎么样，大家是否觉得很神奇？”老师笑着问。

收获，不止 Oracle

“老师，是不是再执行一遍 `alter table range_part_tab exchange partition p8 with table mid_table` 后，`mid_table` 又会为 0，而 `partition(p8)` 又有记录了？” 敬昱有些好奇地问。

“非常好，分区交换能否从普通表交换到分区表，这倒是值得试验一下，让我们继续吧：

```
SQL> alter table range_part_tab exchange partition p8 with table mid_table;
```

表已更改。

```
SQL> select count(*) from range_part_tab partition(p8);
```

```
COUNT(*)
```

```
-----  
      8628
```

```
SQL> select count(*) from mid_table ;
```

```
COUNT(*)
```

```
-----  
        0
```

脚本 4-35 分区交换（从普通表交换到分区表）

敬昱同学，你猜测的对吗？”

“对了！” 敬昱边回答边开心地笑了。

“不过大家这里还要注意 `exchange` 是交换的含义，试验中 `mid_table` 表开始没记录，所以交换过程中总有一边记录为 0。如果 `mid_table` 表初始有记录，比如有 1 条，那 `exchange` 的结果就是一边是 8628 条，另一边是 1 条的交替变换了。” 老师做了补充了。

（3）分区切割你想分就分

“请大家回头复习一下范围分区的语法，其中有一处 `partition p_max values less than (maxvalue)`，老师曾经说过此处表示的是超出这些范围的记录全部落在这个分区中，免得出错，大家还有印象吗？” 老师问。

同学们大多点头。

“大家难道没有什么疑问吗？” 老师又问了。

“老师，2013 年以后的数据都一股脑儿地装到 `P_MAX` 一个分区里，不符合分区表的设计思想啊，是否可以再扩建出多个分区来？” 晶晶忽然有所发现。

“对了，这就是老师今天要给大家说的分区切割，请看下列语句：

```
SQL> alter table range_part_tab split partition p_max at (TO_DATE('2013-02-01', 'YYYY-MM-DD')) into  
(PARTITION p2013_01 ,PARTITION P_MAX);
```

表已更改。

```
SQL> alter table range_part_tab split partition p_max at (TO_DATE('2013-03-01', 'YYYY-MM-DD')) into  
(PARTITION p2013_02 ,PARTITION P_MAX);
```

表已更改。

脚本 4-36 分区切割

操作完后，可以通过以下试验来查看分区切割是否成功，如下：

```
SQL> column segment_name format a20
SQL> column partition_name format a20
SQL> column segment_type format a20
SQL> select segment_name,
  2      partition_name,
  3      segment_type,
  4      bytes / 1024 / 1024 "字节数(M)",
  5      tablespace_name
  6  from user_segments
  7  where segment_name IN('RANGE_PART_TAB');
```

SEGMENT_NAME	PARTITION_NAME	SEGMENT_TYPE	字节数(M)	TABLESPACE_NAME
RANGE_PART_TAB	P1	TABLE PARTITION	6	TBS_LJB
RANGE_PART_TAB	P2	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P3	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P4	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P5	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P6	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P7	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P8	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P9	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P10	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P11	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P12	TABLE PARTITION	3	TBS_LJB
RANGE_PART_TAB	P2013_01	TABLE PARTITION	.0625	TBS_LJB
RANGE_PART_TAB	P2013_02	TABLE PARTITION	.0625	TBS_LJB
RANGE_PART_TAB	P_MAX	TABLE PARTITION	.0625	TBS_LJB

已选择 15 行。

脚本 4-37 观察分区切割情况

可以很明显地看出增加了两个分区，以上操作成功完成了分区表的分区切割的示例，注意如下几点：

- ① 分区切割的三个关键字是 `split`、`at` 和 `into`。
- ② `at` 部分在此处说明了具体的范围，小于某个指定的值。
- ③ `into` 部分说明分区被切割成两个分区，比如 `into (PARTITION p2013_01 ,PARTITION P_MAX)`表示将 `P_MAX` 切割成 `PARTITION p2013_01` 和 `PARTITION P_MAX` 两部分，其中括号里的 `P_MAX` 可以改为新的名字，也可以保留原来的名字。”

（4）分区合并你想合就合

“老师，分区可以切割，还可以合并吗？”小莲脑中忽然闪过这个念头，忍不住打断了老师的话。

“问得好！有分就有合，当然可以合并，要不我们就把刚才切割出来的两个分区再合并回去，你们看如何？”

“好！”同学们回答得同时都睁大眼睛，看老师接下来的操作。

“请同学们看好了，现在我要把刚才切割出来的 PARTITION p2013_01 和 PARTITION p2013_02 都再和 p_max 合并回去，让刚才做的事情复原了，如下：

```
SQL> alter table range_part_tab merge partitions p2013_02, P_MAX INTO PARTITION P_MAX;  
表已更改。  
SQL> alter table range_part_tab merge partitions p2013_01, P_MAX INTO PARTITION P_MAX;  
表已更改。
```

脚本 4-38 分区合并例子

接下来已经 OK 了，我们来验证看看：

```
SQL> column segment_name format a20  
SQL> column partition_name format a20  
SQL> column segment_type format a20  
SQL> select segment_name,  
2      partition_name,  
3      segment_type,  
4      bytes / 1024 / 1024 "字节数(M)",  
5      tablespace_name  
6  from user_segments  
7  where segment_name IN('RANGE_PART_TAB');
```

SEGMENT_NAME	PARTITION_NAME	SEGMENT_TYPE	字节数(M)	TABLESPACE_NAME
RANGE_PART_TAB	P1	TABLE PARTITION	6	TBS_LJB
RANGE_PART_TAB	P2	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P3	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P4	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P5	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P6	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P7	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P8	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P9	TABLE PARTITION	4	TBS_LJB
RANGE_PART_TAB	P10	TABLE PARTITION	5	TBS_LJB
RANGE_PART_TAB	P11	TABLE PARTITION	4	TBS_LJB

RANGE_PART_TAB	P12	TABLE PARTITION	3	TBS_LJB
RANGE_PART_TAB	P_MAX	TABLE PARTITION	.0625	TBS_LJB

已选择 13 行。

脚本 4-39 观察分区合并情况

果然刚才新切割出来的两个分区不见了，又被合并回去了。关于合并的操作，也要注意如下几点：

- ① 分区合并的关键字是 **merge** 和 **into**。
- ② **merge** 后面跟着的是需要合并的两个分区名。
- ③ **into** 部分为合并后的分区名，可以是新的分区名，也可以沿用已存在的分区名。”
- (5) 分区增与删非常简单

“接下来，我们探讨分区操作中的分区的增加与删除，这个比较简单，我们就合并在一起做试验，首先测试增加分区的操作，具体如下：

```
SQL> alter table range_part_tab add partition p2013_01 values less than (TO_DATE('2013-02-01', 'YYYY-MM-DD'));
alter table range_part_tab add partition p2013_01 values less than (TO_DATE('2013-02-01', 'YYYY-MM-DD'))
*
```

第 1 行出现错误：

ORA-14074: 分区界限必须调整为高于最后一个分区界限

脚本 4-40 最后一个分区是 maxvalue，不允许追加，只允许 split

奇怪，怎么出错了？原来在最后一个分区是 less than (maxvalue) 的情况下，是不能追加分区的，只能 SPLIT 分裂。因为追加的分区界限比这个 p_max 还要低，显然不能允许。

不过我们倒是可以改成先试验分区删除，把这个 p_max 给删除了，然后追加自然就没问题了，继续试验如下：

```
SQL> alter table range_part_tab drop partition p_max;
表已更改。
SQL> alter table range_part_tab add partition p2013_01 values less than (TO_DATE('2013-02-01', 'YYYY-MM-DD'));
表已更改。
SQL> alter table range_part_tab add partition p2013_02 values less than (TO_DATE('2013-03-01', 'YYYY-MM-DD'));
表已更改。
```

脚本 4-41 可以删除 maxvalue，进行分区追加

这下试验先分区删除后，分区增加也很容易试验成功了。关于分区增删的操作，要注意如下几点：

- ① 分区增加的关键字是 `add partition`，而分区删除的关键字是 `drop partition`。
- ② 由于 `maxvalue` 分区的存在，无法追加新的分区，必须删除了才可以追加。”

4.2.4.3 分区索引类型简述

1. 全局索引

“说到分区表，其实还有一个不得不提的重要知识点，那就是分区索引，不少分区表设计中正是由于相关索引的错误设计，导致分区表遇到诸多问题。不过由于下一章中老师将会非常详尽地给大家全面讲解索引的知识，所以本章节的分区索引暂且不会做特别详尽的描述，只是抓住一些重点向大家说明一下。

分区表的索引一般可以分成两类，一类是全局索引，另一类就是局部索引，其中全局索引和普通的建索引的方式无异，而局部索引需要增加 `local` 关键字，具体如下：

```
-----以下是对 deal_date 列建全局索引
SQL> create index idx_part_tab_date on range_part_tab(deal_date);
索引已创建。
-----以下是对 area_code 列建一个局部索引
SQL> create index idx_part_tab_area on range_part_tab(area_code) local;
索引已创建。
```

脚本 4-42 全局索引与局部索引

其实全局索引基本上可以理解为普通索引，从如下语句可以看出该索引 `idx_part_tab_date` 的段分配使用情况：

```
SQL> SET LINESIZE 666
SQL> set pagesize 5000
SQL> column segment_name format a20
SQL> column partition_name format a20
SQL> column segment_type format a20
SQL> select segment_name,
2         partition_name,
3         segment_type,
4         bytes / 1024 / 1024 "字节数(M)",
5         tablespace_name
6     from user_segments
7    where segment_name IN('IDX_PART_TAB_DATE');
```

SEGMENT_NAME	PARTITION_NAME	SEGMENT_TYPE	字节数(M)	TABLESPACE_NAME
IDX_PART_TAB_DATE		INDEX	3	TBS_LJB

脚本 4-43 全局索引段分配情况

2. 局部索引

而局部索引明显就有差异了，请同学们继续查看如下结果：

```
SQL> SET LINESIZE 666
SQL> set pagesize 5000
SQL> column segment_name format a20
SQL> column partition_name format a20
SQL> column segment_type format a20
SQL> select segment_name,
2      partition_name,
3      segment_type,
4      bytes / 1024 / 1024 "字节数(M)",
5      tablespace_name
6  from user_segments
7  where segment_name IN('IDX_PART_TAB_AREA');
```

SEGMENT_NAME	PARTITION_NAME	SEGMENT_TYPE	字节数(M)	TABLESPACE_NAME
IDX_PART_TAB_AREA	P1	INDEX PARTITION	.25	TBS_LJB
IDX_PART_TAB_AREA	P2	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P3	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P4	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P5	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P6	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P7	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P8	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P9	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P10	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P11	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P12	INDEX PARTITION	.1875	TBS_LJB
IDX_PART_TAB_AREA	P_MAX	INDEX PARTITION	.0625	TBS_LJB

脚本 4-44 局部索引段分配情况

现在大家应该明白，局部索引其实就是针对各个分区所建的索引。在本例中可以看出，和局部索引相比，全局索引好比一个大索引，而局部索引好比 13 个小索引。”

4.2.4.4 分区表之相关陷阱

“至此，老师将分区表的重点知识基本说完，不过大家要特别注意的是，分区表设计及应用还是要多一些谨慎，一不小心就容易陷入种种误区之中，比如索引无端失效、分区表建了索引后感觉性能不高、建了分区却无法有效应用到分区条件等等。

收获，不止 Oracle

1. 索引缘何频频失效

其中最容易出问题的当属分区表的不当操作导致分区索引失效，这些操作就是前面分区操作这一小节描述的系列动作，这些动作全部都会导致分区索引中的全局索引失效。

以下是查看 `range_part_tab` 表的索引情况，其中 `STATUS` 是 `N/A` 表示是局部索引，需要进一步在 `user_ind_partitions` 中分析其索引的状态，如下：

```
SQL> select index_name, status
2   from user_indexes
3   where index_name in('IDX_PART_TAB_DATE', 'IDX_PART_TAB_AREA');
```

INDEX_NAME	STATUS

IDX_PART_TAB_AREA	N/A
IDX_PART_TAB_DATE	VALID

```
SQL> select index_name, partition_name, status
2   from user_ind_partitions
3   where index_name = 'IDX_PART_TAB_AREA';
```

INDEX_NAME	PARTITION_NAME	STATUS

IDX_PART_TAB_AREA	P1	USABLE
IDX_PART_TAB_AREA	P2	USABLE
IDX_PART_TAB_AREA	P3	USABLE
IDX_PART_TAB_AREA	P4	USABLE
IDX_PART_TAB_AREA	P5	USABLE
IDX_PART_TAB_AREA	P6	USABLE
IDX_PART_TAB_AREA	P7	USABLE
IDX_PART_TAB_AREA	P8	USABLE
IDX_PART_TAB_AREA	P9	USABLE
IDX_PART_TAB_AREA	P10	USABLE
IDX_PART_TAB_AREA	P11	USABLE
IDX_PART_TAB_AREA	P12	USABLE
IDX_PART_TAB_AREA	P2013_01	USABLE
IDX_PART_TAB_AREA	P2013_02	USABLE

已选择 14 行。

脚本 4-45 观察全局和局部索引的状态

现在来看，索引的状态都非常正常，我们先来进行一个分区表的分区 `truncate` 操作，看看索引是否会失效，如下：

```
SQL> select count(*) from range_part_tab partition(p1);
```

```

COUNT(*)
-----
10802
SQL> alter table range_part_tab truncate partition p1;
表被截断。
SQL> select count(*) from range_part_tab partition(p2);
COUNT(*)
-----
0
SQL> select index_name, status
2      from user_indexes
3      where index_name in('IDX_PART_TAB_DATE', 'IDX_PART_TAB_AREA');
INDEX_NAME          STATUS
-----
IDX_PART_TAB_AREA    N/A
IDX_PART_TAB_DATE    UNUSABLE

SQL> select index_name, partition_name, status
2      from user_ind_partitions
3      where index_name = 'IDX_PART_TAB_AREA';

INDEX_NAME          PARTITION_NAME          STATUS
-----
IDX_PART_TAB_AREA    P1                      USABLE
IDX_PART_TAB_AREA    P2                      USABLE
IDX_PART_TAB_AREA    P3                      USABLE
IDX_PART_TAB_AREA    P4                      USABLE
IDX_PART_TAB_AREA    P5                      USABLE
IDX_PART_TAB_AREA    P6                      USABLE
IDX_PART_TAB_AREA    P7                      USABLE
IDX_PART_TAB_AREA    P8                      USABLE
IDX_PART_TAB_AREA    P9                      USABLE
IDX_PART_TAB_AREA    P10                     USABLE
IDX_PART_TAB_AREA    P11                     USABLE
IDX_PART_TAB_AREA    P12                     USABLE
IDX_PART_TAB_AREA    P2013_01                USABLE
IDX_PART_TAB_AREA    P2013_02                USABLE
已选择 14 行。

```

脚本 4-46 做分区 truncate 后全局索引失效，局部索引未失效

我们可以发现，全局索引失效了，而局部索引没有失效，以下不得不进行索引的重建工作，来让索引生效：

```
SQL> alter index IDX_PART_TAB_DATE rebuild;
```

索引已更改。

```
SQL> select index_name, status
2      from user_indexes
3      where index_name = 'IDX_PART_TAB_DATE';
```

INDEX_NAME	STATUS

IDX_PART_TAB_DATE	VALID

脚本 4-47 对失效的全局索引进行重建

其实分区表的分区操作，对局部索引一般都没有影响，但是对全局索引影响比较大。Oracle 在提供这些分区操作时提供了一个很有用的参数 **update global indexes**，可以有效地避免全局索引失效。

其实这个参数的本质动作是在分区操作做完后，暗暗执行了索引重建的工作。现在我们来一组例子，探讨增加这个参数后，全局索引是否还会失效，如下：

```
SQL> select count(*) from range_part_tab partition(p2);
COUNT(*)
```

7881

```
SQL> alter table range_part_tab truncate partition p2 update global indexes;
```

表被截断。

```
SQL> select count(*) from range_part_tab partition(p2);
COUNT(*)
```

0

```
SQL> select index_name, status
2      from user_indexes
3      where index_name = 'IDX_PART_TAB_DATE';
```

INDEX_NAME	STATUS

IDX_PART_TAB_DATE	VALID

脚本 4-48 update global indexes 关键字可避免全局索引失效

其他分区操作，比如分区转移、切割、合并、增删等，也和这个分区 **truncate** 是类似的，都允许增加 **update global indexes** 关键字，从而避免全局索引失效，这里就不一一罗列了。”

2. 有索引效率反更低

“前面的陷阱和索引失效有关，现在我们开始谈的是设计不当产生的性能问题。我们以前面

的组合分区表 `range_list_part_tab` 为例，对其中的 `id` 列建一个局部索引，然后我们对 `norm_tab` 表的 `id` 列也建一个索引，分析看看，针对类似 `select * from norm_tab where id=888` 这样的简单查询，分区表查询是否比普通表更快，如下：

```
SQL> create index idx_range_list_tab_date on range_list_part_tab(id) local;
```

索引已创建。

```
SQL> set autotrace traceonly
```

```
SQL> set linesize 1000
```

```
SQL> select *
```

```
2 from range_list_part_tab
```

```
3 where id=100000;
```

执行计划

```
-----
Plan hash value: 1587118596
```

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT		1	2037	15 (0)	00:00:01		
1	PARTITION RANGE ALL		1	2037	15 (0)	00:00:01	1	13
2	PARTITION LIST ALL		1	2037	15 (0)	00:00:01	1	10
3	TABLE ACCESS BY LOCAL INDEX ROWID	RANGE_LIST_PART_TAB	1	2037	15 (0)	00:00:01	1	130
* 4	INDEX RANGE SCAN	IDX_RANGE_LIST_TAB_DATE	1		14 (0)	00:00:01	1	130

```
-----
Predicate Information (identified by operation id):
```

```
4 - access("ID"=100000)
```

Note

```
-----
- dynamic sampling used for this statement
```

统计信息

```
-----
0 recursive calls
0 db block gets
240 consistent gets
0 physical reads
0 redo size
999 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 4-49 应用分区表的局部索引产生的逻辑读很大

收获，不止 Oracle

接下来是普通表的查询情况，如下：

```
SQL> create index idx_norm_tab_date on norm_tab(id);
索引已创建。
SQL> set autotrace traceonly
SQL> set linesize 1000
SQL> select *
  2   from norm_tab
  3  where id=100000;
```

执行计划

Plan hash value: 2695975962

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	2037	2 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	NORM_TAB	1	2037	2 (0)	00:00:01
* 2	INDEX RANGE SCAN	IDX_NORM_TAB_DATE	1		1 (0)	00:00:01

Predicate Information (identified by operation id):

```
2 - access("ID"=100000)
Note
-----
- dynamic sampling used for this statement
```

统计信息

```
0 recursive calls
0 db block gets
4 consistent gets
0 physical reads
0 redo size
999 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 4-50 应用普通表的普通索引产生的逻辑读很小

同学们，大家觉得谁的性能好？ ”

“老师，显然是普通表查询的性能好，分区表的逻辑读是 **240**，而普通表才 **4**。分区表的 COST 是 15，而普通表的 COST 为 2，为什么会这样啊，真是神奇。”晶晶观察了一会儿后有些吃惊地回答老师。

“这个问题其实涉及索引的一个特点，就是索引的**高度一般比较低**，理解了这个道理，你们就能自然而然地明白为什么性能差别会如此之大了。

关于索引我们在随后一个章节中将会进行详细的描述，学习完索引章节后，这个疑问就迎刃而解了，这里我先给大家保留一个悬念。”老师有些故作神秘。

3. 无法应用分区条件

“大家在分区设计时，往往没有预先规划好如何应用分区，这是很不应该的。我经常看到很多开发人员操作的是分区表，却不用上分区条件，从而无法做到分区消除，这就浪费了分区表的宝贵特性，应该避免出现。

就拿上述例子来说，虽然得出了相同的结果，但是分区表的性能却比普通表差了许多。假如存在这样一种情况：这个查询虽然是针对 id 列，但是加上 `and deal_date >= sysdate-1 and area_code=591` 这样的条件，依然可以等价，那刚才的语句就会飞快了，如下。

```
SQL> select *
      2   from range_list_part_tab
      3   where id=100000
      4   and deal_date >= sysdate-1
      5   and area_code=591;
```

未选定行

Plan hash value: 1587118596

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT		1	2037	4(0)	00:00:01		
1	PARTITION RANGE ALL		1	2037	4(0)	00:00:01	KEY	13
2	PARTITION LIST ALL		1	2037	4(0)	00:00:01	KEY	KEY
3	TABLE ACCESS BY LOCAL INDEX ROWID	RANGE_LIST_PART_TAB	1	2037	4(0)	00:00:01	KEY	KEY
* 4	INDEX RANGE SCAN	IDX_RANGE_LIST_TAB_DATE	45		2(0)	00:00:01	KEY	KEY

Predicate Information (identified by operation id):

4 - access("ID"=100000)

Note

统计信息

0 recursive calls


```
0 db block gets
3 consistent gets
0 physical reads
0 redo size
448 bytes sent via SQL*Net to client
389 bytes received via SQL*Net from client
1 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
0 rows processed
```

脚本 4-51 分区表设计要考虑在语句中有效用到分区条件，有无分区条件差别巨大

这里性能显然提升了许多，逻辑读从原来的 240 瞬间降低到 3，其实原因在于之前遍历了所有的分区，现在只访问指定的分区。

只可惜，我这里加的条件只是一个假如，并不能真正加上去。开发人员和设计人员在设计分区表时，要务必了解如何应用上分区条件，从而有效地做到分区消除，这是非常重要的，大家记住了吗？”

“记住了！”

4.2.5 有趣的索引组织表

“大家还记得之前说过普通堆表操作的不足之处吗，比如 `select * from t where id=1` 之类的查询，`id` 列有索引，如果是普通的表，需先从索引中获取 `rowid`，然后定位到表中，获取 `id` 以外的其他列的动作，这就是回表。

如果查询列含索引列以外的列，回表就不可避免，这个大家明白吧？”

台下同学们纷纷点头。

“现在老师要变一个魔术，要让诸如 `select * from t where id=1` 这类的查询可以不回表，大家信不信？”老师笑着问。

同学们没有应答，不过满怀期待。

“请同学们先看一组试验，首先是准备工作，分别建普通表和索引组织表并插入部分数据，大家特别注意，其中的 `organization index` 关键字就是索引组织表的语法，索引组织表必须有主键。具体如下：

```
SQL> drop table heap_addresses purge;
表已删除。
SQL> drop table iot_addresses purge;
表已删除。
SQL> create table heap_addresses
```

```

2  (empno    number(10),
3   addr_type varchar2(10),
4   street   varchar2(10),
5   city     varchar2(10),
6   state    varchar2(2),
7   zip      number,
8   primary key (empno)
9  )
10 /

```

表已创建。

```
SQL> create table iot_addresses
```

```

2  (empno    number(10),
3   addr_type varchar2(10),
4   street   varchar2(10),
5   city     varchar2(10),
6   state    varchar2(2),
7   zip      number,
8   primary key (empno)
9  )
10 organization index
11 /

```

表已创建。

```
SQL> insert into heap_addresses
```

```

2  select object_id,'WORK','123street','washington','DC',20123
3  from all_objects;

```

已创建 71612 行。

```
SQL> insert into iot_addresses
```

```

2  select object_id,'WORK','123street','washington','DC',20123
3  from all_objects;

```

已创建 71612 行。

```
SQL> commit;
```

提交完成

脚本 4-52 分别建索引组织表和普通表进行试验`

接下来对两表进行一个简单性能查询比较，如下：

```
SQL> set linesize 1000
```

```
SQL> set autotrace traceonly
```

```
SQL> select *
```

```

2  from heap_addresses

```

收获，不止 Oracle

```
3  where empno=22;
```

执行计划

Plan hash value: 3326521127

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	50	1 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	HEAP_ADDRESSES	1	50	1 (0)	00:00:01
* 2	INDEX UNIQUE SCAN	SYS_C0011198	1		1 (0)	00:00:01

Predicate Information (identified by operation id):

2 - access("EMPNO"=22)

统计信息

```
0 recursive calls
0 db block gets
3 consistent gets
0 physical reads
0 redo size
659 bytes sent via SQL*Net to client
404 bytes received via SQL*Net from client
1 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

SQL> select *

```
2  from iot address
3  where empno=22;
```

执行计划

Plan hash value: 2472252982

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	50	1 (0)	00:00:01
* 1	INDEX UNIQUE SCAN	SYS_IOT_TOP_74946	1	50	1 (0)	00:00:01

Predicate Information (identified by operation id):

1 - access("EMPNO"=22)

统计信息

```

0 recursive calls
0 db block gets
2 consistent gets
0 physical reads
0 redo size
751 bytes sent via SQL*Net to client
415 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 4-53 分别比较索引组织表和普通表的查询性能

同学们，看看这个试验结果，大家有什么发现吗？”老师问。

“索引组织表的逻辑读是 2 而普通表的逻辑读是 3，另外普通表读取主键索引后，为了获取索引列以外的列信息，产生了回表 **TABLE ACCESS BY INDEX ROWID**，而索引组织表没有，我想这一定就是该试验中索引组织表性能更高的原因所在。”晶晶回答得很自信。

“说得非常好，索引组织表最大的特点就是，表就是索引，索引就是表，这是一种很特别的设计，所以无须访问表。不过这种设计的表的更新要比普通表开销更大。因为表要和索引一样有序地排列，更新负担将会非常严重。因此这种设计一般适用在很少更新、频繁读的应用场合，比如地区配置表，这种表数据一般很少变动，却大量读取。

关于索引组织表，老师这里不做过多的说明，只是告诉大家有这么一种东西，可以解决普通表索引读大多需要回表的缺陷，但是却要小心使用，尤其是在表频繁更新时更需要慎之又慎。

大家如果要深入了解具体的知识，可以详细阅读 Oracle 的官方文档，本节课程，以及下一小节的簇表介绍，主要是抛砖引玉，让大家开阔一下知识视野。

实际项目中索引组织表和簇表应用的场合相对较少，但是如果选择合适的时机用上，也会成为一件秘密武器，一把飞刀。”

“飞刀？梁老师这是给我们制造悬念啊，看来我抽空得去好好了解一下。”小莲心中暗想。

4.2.6 簇表的介绍及应用

“老师说过普通表还有一点缺陷，就是 ORDER BY 语句中的排序不可避免，大家还记得吗？”

“记得。”

“实际上有序簇表可以避免排序，大家看看老师构造的一个试验，如下：

收获，不止 Oracle

```
SQL> Drop table cust_orders;
```

表已删除。

```
SQL> Drop cluster shc;
```

簇已删除。

```
SQL>
```

```
SQL> CREATE CLUSTER shc
```

```
 2 (
 3   cust_id      NUMBER,
 4   order_dt     timestamp SORT
 5 )
 6 HASHKEYS 10000
 7 HASH IS cust_id
 8 SIZE 8192
 9 /
```

簇已创建。

```
SQL> CREATE TABLE cust_orders
```

```
 2 (   cust_id      number,
 3     order_dt     timestamp SORT,
 4     order_number  number,
 5     username     varchar2(30),
 6     ship_addr    number,
 7     bill_addr    number,
 8     invoice_num   number
 9 )
10 CLUSTER shc ( cust_id, order_dt )
11 /
```

表已创建。

```
SQL> ---开始执行分析
```

```
SQL> set autotrace traceonly explain
```

```
SQL> variable x number
```

```
SQL> select cust_id, order_dt, order_number
```

```
 2   from cust_orders
 3   where cust_id = :x
 4   order by order_dt;
```

执行计划

Plan hash value: 465084913

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
----	-----------	------	------	-------	-------------

```

-----
| 0 | SELECT STATEMENT |          | 1 | 39 | 0 (0) |
|* 1 | TABLE ACCESS HASH| CUST_ORDERS | 1 | 39 |        |
-----

```

Predicate Information (identified by operation id):

```

-----
1 - access("CUST_ID"=TO_NUMBER(:X))

```

Note

```

-----
- dynamic sampling used for this statement (level=2)

```

脚本 4-54 簇表设计好，可避免排序

这个例子大家暂且记着，老师也不做过多讨论，关于避免排序，还有另外一种方法，也是更常见的方法：排序列正好是索引列时，可避免排序。关于索引避免排序这个知识，老师将会在下一章与索引相关的课程中给大家做详细的介绍。

簇表和索引组织表一样，由于结构的特殊导致更新操作开销非常大，所以也需要谨慎使用。更多的知识细节，可以从官方文档中学习。

此外大家回去把老师讲的这个章节好好体会一下，看看是否有机会应用到相关工作中，下一次课程开始时我们先分享大家的心得体会。今天的课程到此结束，谢谢大家！”

4.3 理解表设计的你成为项目组英雄

“同学们，很高兴又和大家见面了，最近一周时间大家可有工作中应用到表设计的心得体会来和大家分享呢？”老师满怀期待地望着大家。

“老师，您介绍的全局临时表特性让我解决了项目组困扰已久的问题，我简直成为项目组的英雄了！”晶晶回答得有些激动。

“哦，真好，继续说。”老师心里很是开心。

“我们当前系统的架构设计有一部分是这样的：

- ① 原始数据来源是一些通过 FTP 程序实时传送到我们指定平台的文件。
- ② 这些文件由我们的后台应用程序来实时处理，即依次读取文件里的各行内容并写入到数据库指定表 T0 中。
- ③ 数据库中的这张 T0 表是原始数据，只能是中间处理环节而非最终数据，需要对这些原始数据进行筛选过滤，最终插入到指定的 5 张表 T1、T2、T3、T4、T5 中，这 5 张表才是最终可用数据。
- ④ 完成通过特定逻辑插入 5 张目标表记录后，原始记录 T0 表内的数据就要被清空。

- ⑤ 接下来前台应用程序通过一定业务逻辑读取这 5 张表的数据，将结果展现在前台页面上。
- ⑥ 每当有新的文件被送到我们指定平台处理时，T0 表又将会有数据，新的循环再次开始。

当前这个架构设计遇到一些性能瓶颈，由于程序处理比较缓慢，经常出现由于处理速度缓慢导致 FTP 过来的文件被大量积压的情况，引发了不少投诉，客户对此向我们多次提出优化要求，而我们一直努力却收效甚微，大家压力都很大。

我首先想到老师说的少做事原则，我想到在这个业务场景下，步骤 4 每次都需要完成插入目标表后 delete 清空 T0 原始表，这个动作如果能不做不就能提高性能吗，这就是梁老师一直强调的少做事原则。我很自然地联想到了您说全局临时表，至少在清空原始表的这个环节上少做了一步，只要 COMMIT 或者退出 SESSION，该表记录就自然清空了。

而从插入数据来看，T0 表如果是全局临时表产生的，日志也会少得多，插入本身的速度也会快得多，这显然又是一个改进。此外在事务控制的安全保障下，全局临时表的数据也不怕丢失，因为和后面的目标表在同一事务中，要么一起成功，要么一起失败重来，很安全的。

后来在我的建议下，项目组同意将 T0 中间表从普通表的类型改造为全局临时表，经过测试发现整个处理环节速度提升了近 3 倍，基本解决了高峰时候文件积压的问题。

不过最让人激动的事情并不限于此，我忽然想到梁老师说过全局临时表针对不同的会话独立，这让我想到了原先不方便实施的并行处理，现在已经没有困难了。

原先程序在处理文件时并没有考虑并行，主要是因为这个顾虑：比如并行 5 个进程开始处理，要插入到 5 张不同的中间表中，如 T01 到 T05，数据库中对应的程序包也要从 1 个变成 5 个，差别就只是 T01 和 T02 等的差别，这样很不利于扩展，所以当时在设计的时候放弃了并行的想法。

现在就简单了，无论后台应用程序起多少个进程，到数据模块，只要是不同的 SESSION，看到的就是各自独立的记录，对应同一个数据库程序包即可。并行一下子变得非常容易了。接下来我把这个建议和项目组分享后，得到了项目组的一致认可，我们将后台程序的并行度设置为 8（一般可根据 CPU 个数来设定）后，整个处理能力从原先的提升 3 倍到提升 10 倍。

然后我就在项目组出名了，成英雄了。”晶晶说到这里，忍不住自己笑出声来。

老师忍不住鼓起掌来，带动了台下掌声一片。

第 5 章



惊叹，索引天地妙不可言

5.1 看似简单无趣的索引知识

索引相关的相关知识汇总如图 5-1 所示。

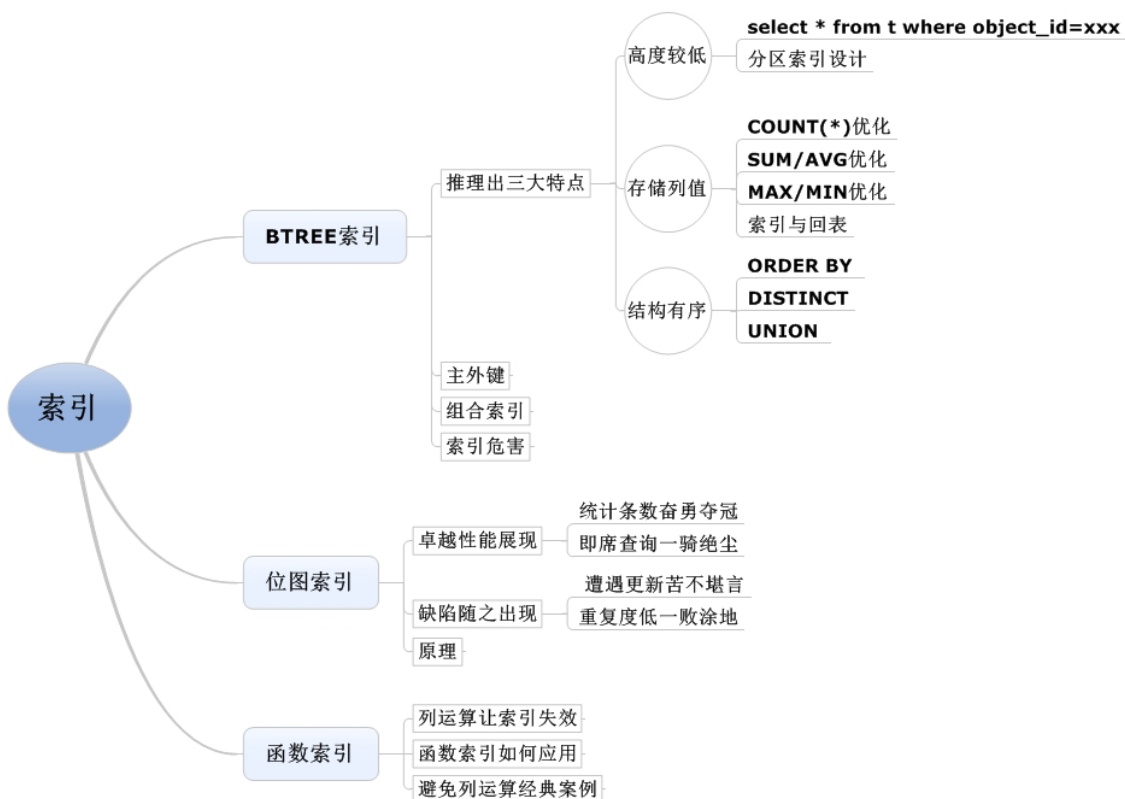


图 5-1 索引

5.2 索引探秘从小余缉凶拉开帷幕

5.2.1 BTREE 索引的精彩世界

5.2.1.1 BTREE 索引结构图展现

“大家好，我们又见面了。”梁老师笑吟吟地出现在讲台上，“前面的课上我们总结过学习路线图，学习了 Oracle 的物理、逻辑体系等重要知识，并将这些与具体的应用落地结合在一起，大家没有忘记吧？”

“没有！”从整齐的回答中可以看出，大家还是学得挺用心的。

“很好，有了逻辑体系结构的基础，我就可以方便地描述索引相关知识了。索引和表一样，都是前面描述的逻辑体系结构中的段的一种，当你建一个 T 表，就产生一个 T 表的表 SEGMENT，当你在 T 表的某些列上建索引 IDX_T，就产生一个 IDX_T 的索引 SEGMENT。索引是建在表的具体列上，其存在的目的是让表的查询更快，效率更高。表记录丢失关乎生死，而索引丢失只需重建即可，似乎听起来索引只是表的一个附属产品，可有可无。

但是在我看来，索引却是数据库学习中最实用的技术之一，谁能深刻地理解和掌握索引的知识，谁就能在数据库相关工作中发挥巨大的作用。了解索引之前，我们需要先了解索引结构长啥样。

先看看图 5-2 的索引结构图。

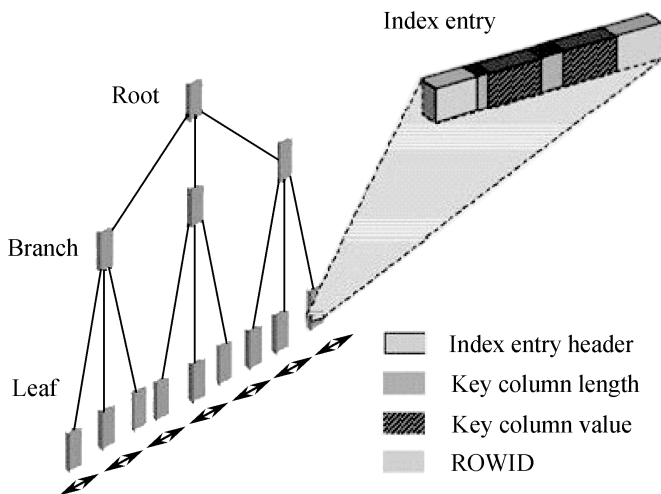


图 5-2 索引结构图

以上结构图说明索引是由 Root（根块），Branch（茎块）和 Leaf（叶子块）三部分组成的，其中 Leaf（叶子块）主要存储了 key column value（索引列具体值），以及能具体定位到数据块所在位置的 rowid（注意区分索引块和数据块）。

我们来具体说说某个 Oracle 的索引查询吧，如：select * from t where id=12; 该 test 表的记录有 10050 条，而 id=12 仅返回 1 条，test 表的 id 列建了一个索引，索引是如何快速检索到数据的呢，接下来分析这个索引查询示例图如图 5-3 所示。

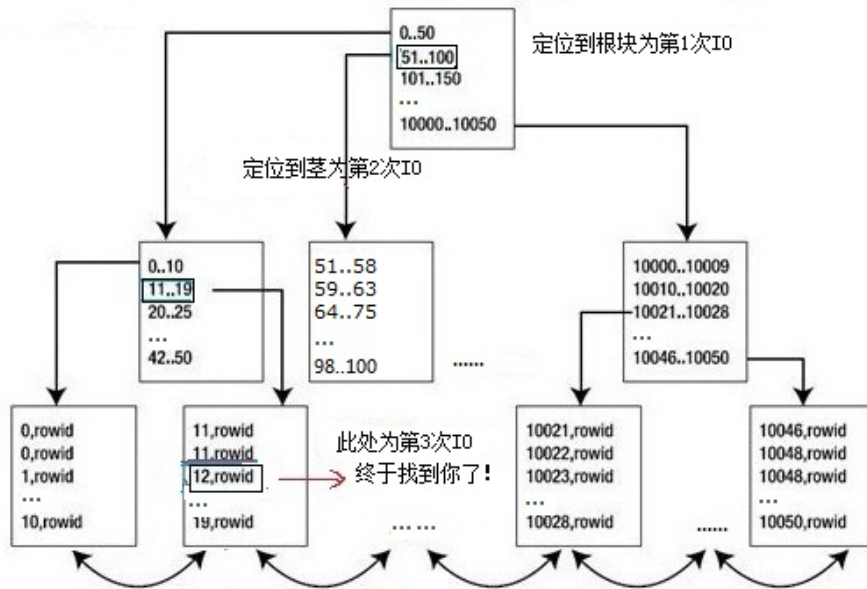


图 5-3 索引查询示例图

通过分析该图片，可以大致理解，定位到 select * from t where id=12 大致只需要 3 个 IO（此处我只是举个例子，1 万多条记录实际情况可能只需要 2 个 IO，这个和索引的高度有关，后续会深入讨论）。

首先查询定位到索引的根部，这里是第 1 次 IO；接下来根据根块的数据分布，定位到索引的茎部（查询到 12 的值的大致范围，11..19 的部分），这是第 2 次 IO；然后定位到叶子块，找到 id=12 的部分，此处为第 3 次 IO。假设 Oracle 在全表扫描记录，遍历所有的数据块，IO 的数量必然将大大超过 3 次。有了这个索引，Oracle 只会去扫描部分索引块，而非全部，少做事，必然能大大提升性能。

至此，索引的结构说完了，索引如何快速检索也说完了，大家听懂了吗？”

“梁老师，我看您只说访问索引块就可以完成 select * from t where id=12 的查询了，根据 id 列上的索引来查询数据只需要访问索引块，不需要访问数据块吗？”小莲有些疑惑地问梁老师。

“小莲同学一针见血地指出了梁老师的错误或者说是描述不完整的部分，非常好!”梁老师笑着说，“我每次给别人培训时都故意说到前面就结束了，然后等同学来发现这个漏洞，但是很少有人质疑，小莲同学非常认真，值得表扬。

这里的语句是 `select * from t where id=12`，这个*表示要展现 t 表的所有字段，显然只访问索引是不可能包含表的所有字段的，因为该索引只是对 id 列建索引，也就存储了 id 列的信息而已。因此上述查询访问完索引块后，必然要再访问数据块，比较快捷的方法是用索引块存储的 rowid 来快速检索数据块（具体在后续章节会描述），由此证明我之前假想的 3 次 IO 是错误的，理应增加一次从索引块到数据块获取各个列信息的检索动作，至少是 4 次 IO 才对。

现在大家都没疑问了吧？”

小莲和同学们都纷纷点头，大家都觉得没疑问了。

“既然大家都没疑问了，梁老师想问大家一个问题，什么情况下查询可以只访问索引而不访问表呢？”

“梁老师，我知道了，如果查询只检索索引列信息，就可以不访问表了，比如查询改成 `select id from t where id=12` 时就是这种情况。”晶晶起身回答。

“非常正确，虽然 `select id from t where id=12` 的写法会让人觉得有些奇怪，但是从回答这个问题的角度来看，却是百分之百正确的回答！没理解的同学请仔细回味我说过的这句话：

其中 Leaf（叶子块）主要存储了 key column value（索引列具体值）以及能具体定位到数据块所在位置的 rowid（注意区分索引块和数据块）。

这句话很经典哦，大家要牢记在心，我相信小莲和晶晶已经完全明白这句话的意思了，对不对啊两位同学？”

两位同学都不好意思地笑了。

5.2.1.2 到底是物理还是逻辑结构

“同学们，`select * from t where id=12` 这个语句在索引中检索时的 3 次 IO 描述大家还记忆犹新吧，大家说说看这个索引查询示例图描述的是一种逻辑结构还是物理结构呢？”

我再解释一下，物理结构可以理解为真正存在根、茎、叶的 BLOCK。而逻辑结构可以理解为并没有存在根、茎的 BLOCK，只是一种指针或者说一种内部算法。

有奖竞猜，大家踊跃发言。”

“逻辑结构!”台下几乎没有听到有人说是物理结构的。

“换句话说就是大家都认为不存在真正的根、茎、叶的块组成了这些对象，是这样吗？”

“是!”虽然大家都回答是，但是可以看出来不少同学们还是有些不敢确定。

“那答案到底是什么呢？老师不着急公布答案，大家跟我一起做一个试验吧，不过这次试验与之前的风格不同，老师并不执行任何脚本，而是通过一系列有趣的看图说话让大家跟梁老师一起想象，大家要打起精神认真听。

有一张 test 表，该表有大致 name (varchar2(20)), id (number), height (number), age (number) 等字段。当前该表有记录，我们要对 test 表的 id 列建索引，create index idx_id on test(id)；执行这个动作到完毕，将会发生一系列什么事情呢？来个系列看图说话吧。

1. 要建索引先排序

未建索引的 test 表大致记录如图 5-4 所示，NULL 表示该字段为空值，此外省略号表示略去不显示内容。注意 rowid 伪列，这个是每一行的唯一标记，每一行的 rowid 值绝对不重复，可定位到行的记录在数据库中的位置（具体在后续的章节中详细介绍）。

name	age	height	id	伪列
小余	100000	rowid
老张	5	rowid
老王	3	rowid
小马	1	rowid
大刘	4	rowid
小明		rowid
小黄	2	rowid
老李	7	rowid
.....
.....	6

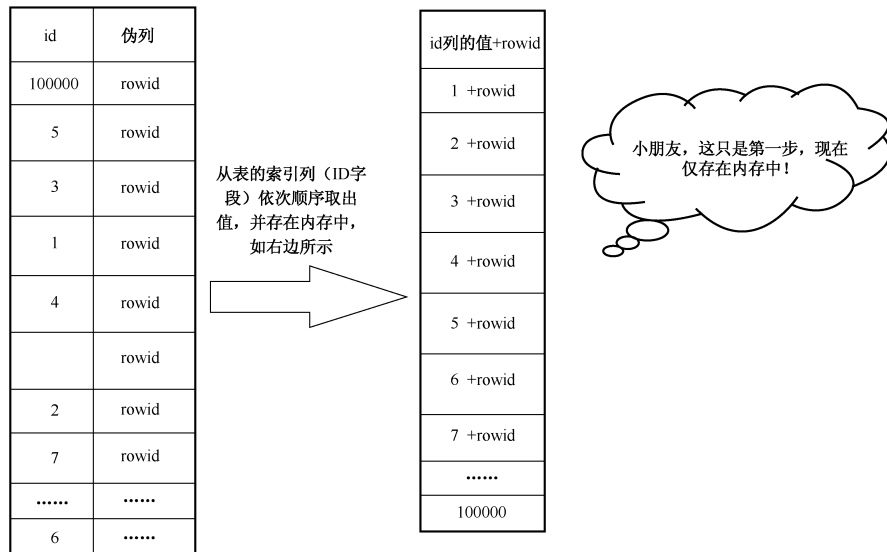
图 5-4 未建索引的 test 表

建索引后，先从 test 表的 id 列的值顺序取出数据放在内存中（这里需注意，除了 id 列的值外，还要注意到取该列的值的同时，该行的 rowid 也被一并取出），如图 5-5 所示。

2. 列值入块成索引

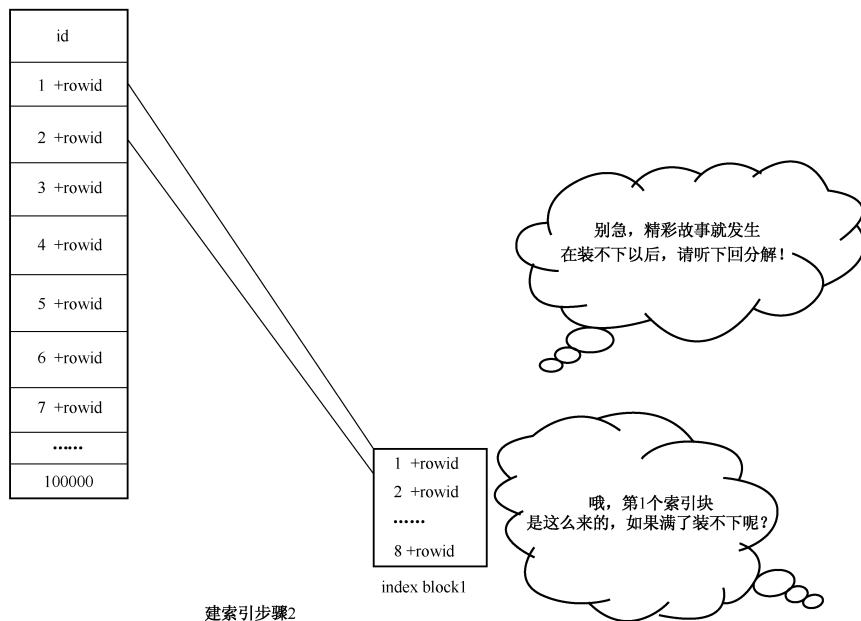
依次将内存中的顺序存放的列的值和对应的 rowid 存进 Oracle 空闲的 BLOCK 中，形成了索引块，具体如图 5-6 所示。

create index idx_id on test(id);



建索引步骤1
(排序从小到大取索引列记录及行rowid进内存)

图 5-5 建索引步骤 1



建索引步骤2
(依次将顺序取得的索引列值以及rowid存进ORACLE的BLOCK中，如图中index block1所示)

图 5-6 建索引步骤 2

3. 填满一块接一块

随着索引列的值的不断插入，index block1(L1)很快就被插满了，比如接下来取出的 id=9 的记录无法插入 index block1(L1)中，就只有插入到新的 Oracle 块中，如图 5-7 所示 index block2(L2)。与此同时，发生了一件非常重要的事情，就是新写数据到另一个块 index block3(B1)，这是为啥呢？原来 L1 和 L2 平起平坐，谁都不服谁，打起来了，不得了了，无组织无纪律哪能行，赶紧得有人管啊，于是 index block3(B1)就担负起管理的角色，这个 BLOCK 记录了 L1 和 L2 的信息，并不记录具体的索引列的键值，目前只占用了 B1 一点点空间。具体细节如图 5-7 所示。

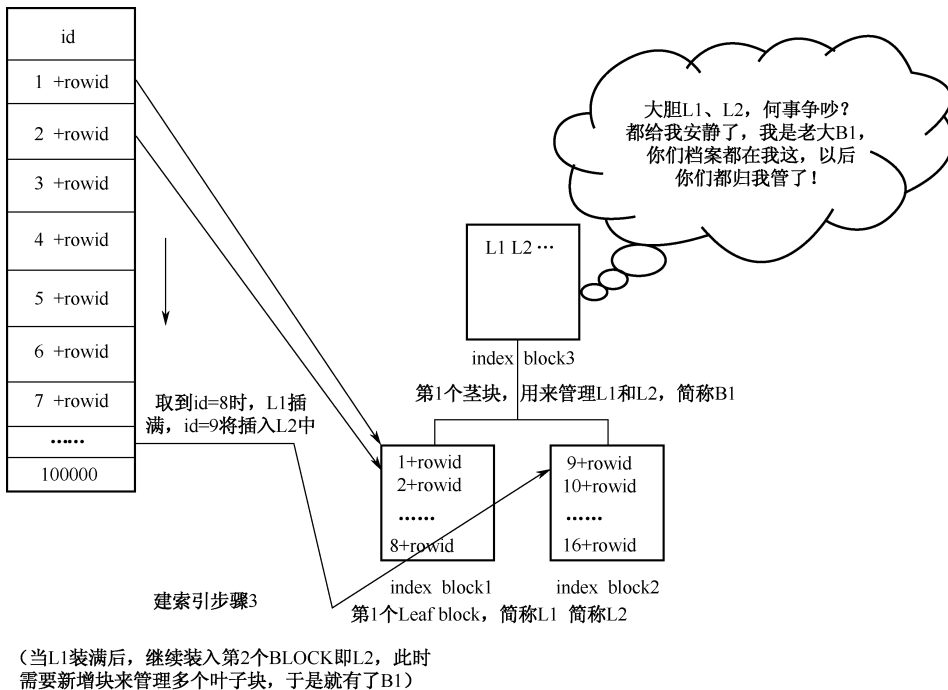


图 5-7 建索引步骤 3

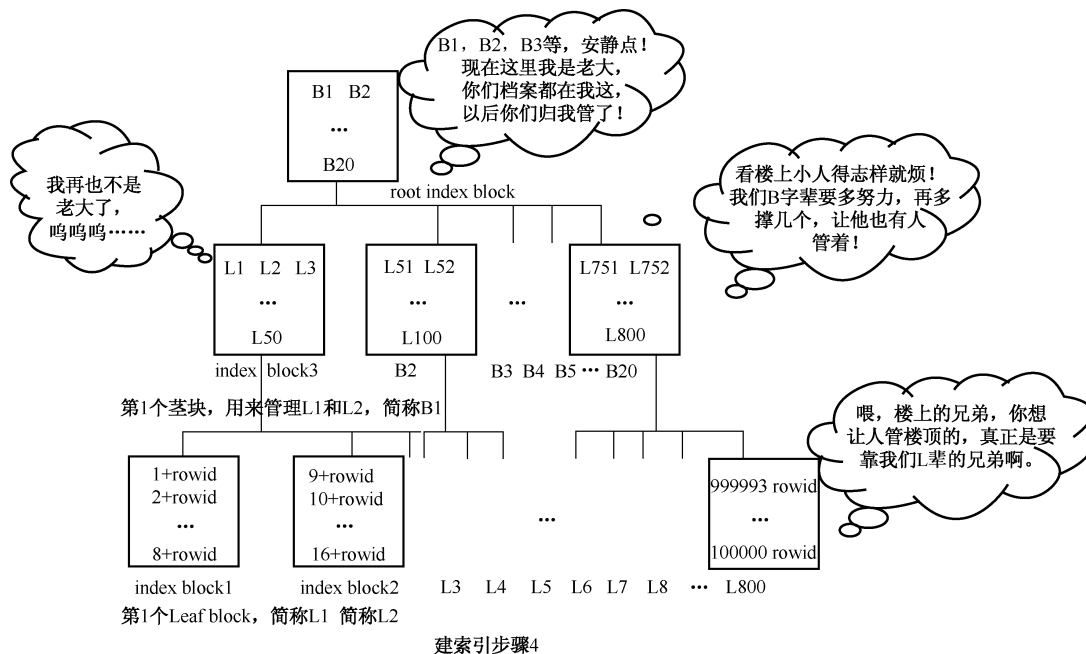
4. 同级两块需人管

随着叶子块的不断增加，B1 块中虽然仅是存放叶子块的标记，但也挡不住量大，最终也容纳不下了。咋办？接着装呗，到下一个块 B2 块去寻找空间容纳。这时 B1 和 B2 也平起平坐了，谁都不服谁，又要打起来了！

‘B1、B2，你们干吗，老实点！’一个威严的声音传来，最上层的 root 根块诞生了，并且有效地管住了这两个小诸侯。B1、B2 唯唯诺诺，老大好！

后续还会出现 B3、B4……如果有一天，这些 Bn 把 root 块撑满了，root 块就不是 root 块了，他也要被人管着，他的上面就又有领导了，不再是高高在上，一统天下了……

具体如图 5-8 所示，至此，你们弄清楚索引的结构了吗？



随着叶子块的不不断增加，B1块也装不下，于是装入B2，新的故事就从这里开始了……

图 5-8 建索引步骤 4

好了同学们，现在你们告诉我，索引查询示例图描述的是一种逻辑结构还是物理结构呢，真有索引的根茎叶块吗？”

同学们都笑了，这下答案明摆着了，还需要回答吗？

小莲心中暗自赞叹梁老师的这种描述方式，这下完全弄清楚索引的来龙去脉了。

5.2.1.3 索引结构三大重要特点

“同学们，通过前面的描述，索引就已经说完了，今天的课程就可以到此结束了，大家回去自由活动吧。”

课程才开始不久，就结束了？同学们一愣，很快明白过来梁老师是逗大家，大家都笑了。

“别笑，如果我们来探索索引的基本原理结构，老师真的是上完了，不开玩笑的。”梁老师一本正经地说。

同学们有些蒙了。

“看来大家都不想提早回家啊，估计是怕老师早退被公司发现后开除了，大家真好。”梁老师笑着说，“但是梁老师真是把索引的基本原理、知识点都说完了，没话说了。看来必须再瞎扯一

下打发时间了。”

台下笑声一片，小莲心里很期待，因为他知道根据梁老师的风格，最精彩的部分马上要出来了。

“大家仔细观察刚才的建索引步骤4中出现的索引结构图，说说有什么特点吧？”

1. 索引高度较低

台下安静了下来，大家认真地思考着，不过半天没人答复。

“那大家按从下往上的角度再看看这个建索引步骤4演变出来的结构图5-9吧。

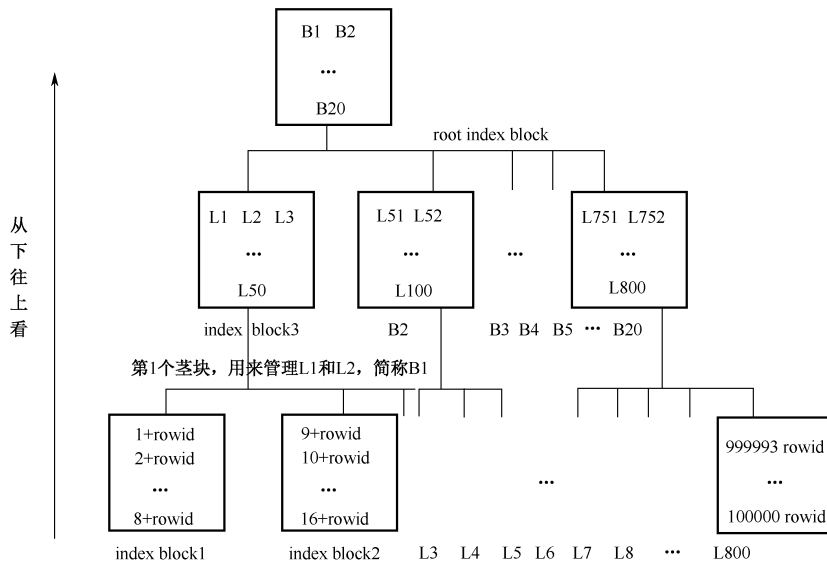


图 5-9 索引从下往上看

有什么发现吗，同学们。”

“梁老师，您让我从下往上看，让我想到这个‘金字塔’应该不容易非常高。”敬昱起身回答。

“为什么呢？”梁老师问。

“最底层的叶子块 index block 因为装具体的数据，所以比较容易被填满，特别是对长度很长的列建索引时更是如此。但是第1层之上的第2层的 index block 就很不不容易装满了吧，因为第2层只是装第1层的指针而已，而第3层是装第2层的 index block 的指针，更不容易了……”

如此说来，这个‘金字塔’如果有好多层，表的数据量应该超级大才对。”敬昱胸有成竹地解释。

“说得非常好！梁老师见过的表最大的有 500G 一张，记录有几百亿条，但是该表上某列索引的高度才不过 6 层而已。”

台下同学们传来一阵惊呼。

2. 索引存储列值

“很好，大家想明白了，还有什么其他特点呢？”

看得出台下大家都在认真思考，不过依然没人回答。

“大家没想明白，没关系，请认真看图 5-10 这个结构图，请仔细往索引叶子块里瞧瞧。

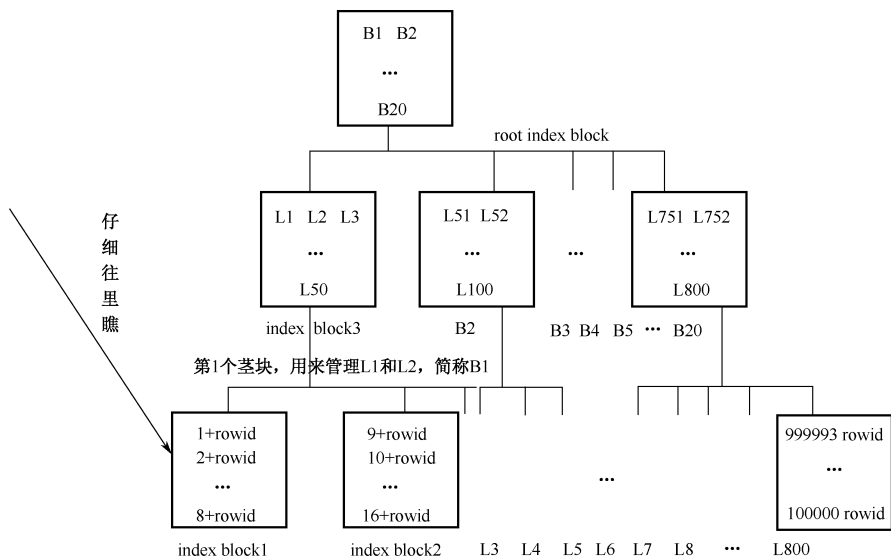


图 5-10 索引仔细往里瞧

有什么发现吗？”

“梁老师，索引存储了表的索引所在列的具体信息，这算是一个特点吗？”晶晶抢先回答。

“算，当然算了！”老说回答，“不过除了存储了索引列的具体内容，最好补充一下还包含了标记定位行数据在数据库中位置的 rowid 取值。”

3. 索引本身有序

“敬昱和晶晶同学回答得非常好，大家思考一下还有什么其他特点吗？”

台下依旧安静，只见大家认真思考，不见大家举手回答。

“梁老师继续启发一下大家，请认真看索引结构图（如图 5-11 所示），此次我们继续换一个角度从左到右看。

又有什么新规律发现了吗？”梁老师微笑着继续引导同学思考。

“梁老师，我知道了，索引是顺序从表里取出数据，再顺序插入到块里形成索引块的，所以说索引块是有序的，比如您的例子里，就是从左到右依次增大的取值。这应该是索引的一个特点吧。”小莲起身回答。

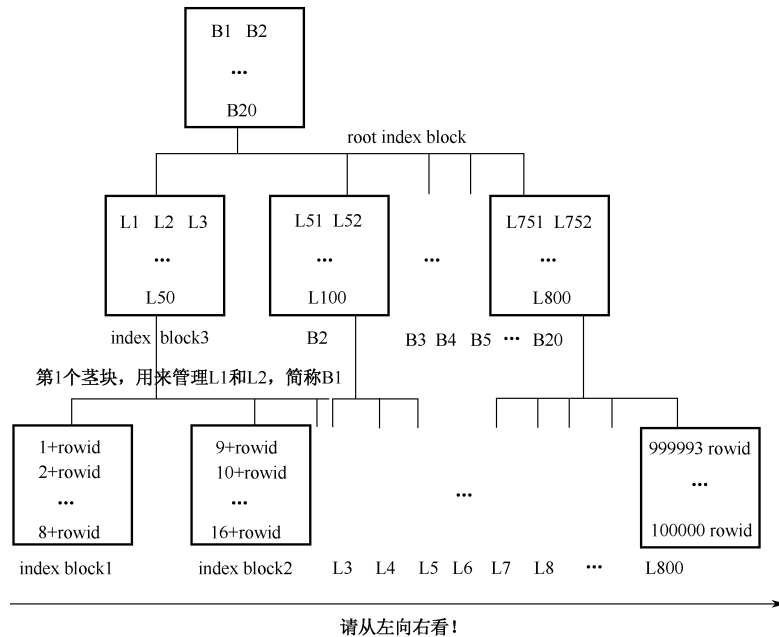


图 5-11 索引从左向右看

“好了，索引的三大特点出来了，如下：

- ① 索引树的高度一般都比较低。
- ② 索引由索引列存储的值及 **rowid** 组成。
- ③ 索引本身是有序的。

这三个特点是在梁老师启发下同学们自己思考出来的，应该印象很深刻，都记住了吗？”

“记住了！”大家齐声回答。

“梁老师费尽心思让大家自己琢磨出这三大特点，可是用心良苦的，希望你们要记牢了。你们无论在官方文档还是网络上的各类文章中，都不会看到像梁老师这样总结并强调出这三个特点，并在后续中巧妙地不断应用。这些特点将会给数据库的开发设计优化的相关工作带来意想不到的好处，后续大家会体会得越来越深刻的。”

“哦，这三个特点的应用有这么神奇吗？”小莲虽说有些疑惑，但更多的却是期待。

5.2.1.4 插播小余缉凶精彩故事

“大家听我描述索引这么久了，可能有一些累了吧，我想让大家调节一下，暂且先不说索引了。因为现在大学一般都设有标准 SQL 学习的相关课程，所以你们中间即便是刚毕业的同学，也大多接触过 SQL，大家应该对 SQL 都很熟悉。

现在我和大家讨论另外一个问题，说说看你们平时写的频率最高的 SQL 单表写法有哪些，请

注意我说的单表两字，复杂的多表连接的写法我们暂且不提，请大家踊跃发言。”梁老师问。

“梁老师，我最常用的就是 `select * from tab where col1='abc'` 此类的查询。”曾祥同学第一个起身回答。

“梁老师，我还比较经常查询多个条件的，比如带地区查找，如 `select * from tab where col1='abc' and area_code=591` 等此类的语句。”敬昱想了想也起身回答。

“梁老师，我补充一下，我经常还有带函数的查询，比如刚才我说的语句加上小写的函数 `lower`，类似 `select * from tab where col1=lower('abc')` 的写法。”曾祥起身做了补充回答。

“很好，还有吗？”梁老师继续问。

“梁老师，我比较经常接触统计报表相关应用，我用得最多的是统计类的语句，比如 `select count(*) from tab` 和 `select max(col) from tab` 之类的语句。”晶晶起身回答。

“我还经常用到聚合分组的语句，比如 `select col1,count(*) from t group by col1` 之类的聚合语句。”小莲也举手回答。

“说得非常好，大家都说出了自己平时使用频率最高的，但是又特别简单的 SQL 语句，相信在座的同学们没人看不懂同学们说的这些语句。那还有没有特别常见的 SQL 写法呢？梁老师继续引导。

“排序，项目组经常需要顺序展现某些记录，我经常要写带 `order by` 的语句，类似 `select * from tab where...order by id` 等写法。”小莲再次举手回答。

“好了，我看大家也说得差不多了，现在我想问大家一个问题，你们平时写这些语句时，有想过执行的效率问题吗？有没有想过利用索引或者其他什么方式，来优化这些 SQL？”梁老师问。

“梁老师，这些语句太简单常见了，能有什么效率问题啊？”敬昱疑惑地说。

“梁老师，项目忙得都做不完了，能实现就不错了，我根本没时间想效率，而且您不是让我们说最简单的 SQL 吗，都最简单的 SQL 了，还怎么优化呢？”小余连珠炮似地说。

“梁老师，我觉得曾祥和敬昱描述的类似 `select * from tab where col1='abc'` 或者 `lower(col1)='abc'` 以及 `where col1='abc' and area_code=591` 组合条件查询，这些倒是可以优化，应该就是建索引吧。

其他的语句无论 `COUNT(*)` 还是 `MAX`，还是 `ORDER BY`，都是从头遍历到尾才知道答案的，这没法用索引来优化吧，索引只适合返回少数记录情况下的优化，遍历所有记录时也不适合使用吧。”小莲略微肯定地回答。

“有自己的想法就应该表扬，小莲同学说得很好，虽然错得一塌糊涂。”梁老师笑着说。

听梁老师这么一说，同学们也都笑成一片。

“准确地说，刚才同学们描述到的所有语句，包括聚合查询，排序在内，全部可以考虑用索引来优化，而且往往效果非常明显。我前面上课时经常提及学习某某知识有啥意义，和不知道这些知识的人有啥差别，在索引这个章节可谓差别明显了，深刻了解了索引知识，刚才同学提及的

所有 SQL，都可以优化！

我个人觉得索引是一个特别有趣的章节，每次我给学员上这个章节时，就会有一种热血沸腾的感觉，因为看上去一个平淡无奇的索引结构，只要我们透过现象看本质，通过推理的方式，居然就可以演绎出诸多优化的经典手法，而这些优化手法却是面向最普通常见的 SQL 的，梁老师将会后面的章节中进行深入描述。

不过在这之前，梁老师想先给大家说一个神探小余的故事，如何？”

“好啊！”台下一片欢呼。

“话说若干年后，小余警校毕业，成长成为一名侦探。有一次，他根据线索找到了嫌疑犯的住所后带队破门而入，发现罪犯并不在里面，而窗门大开，正当大家急于离开继续寻找线索抓捕时，稍作观察的小余立即告知大家罪犯并没离开，让部分警员把守大门，其余人在房间内仔细搜捕，最后终于在房间的一个暗箱里找到了隐藏的罪犯。

小余怎么这么神呢，他靠的就是细致观察加上推理分析。他发现了客厅茶几上泡着一杯茶水，同时他注意到房间的气温比较低。接下来他将手指感受到的茶水的温度和当前房间气温判断，推断出罪犯的茶水应该是刚刚泡下不超过 1 分钟。而这次警员追捕是在外面埋伏已久，从各个方向同时包围上来的，1 分钟之内逃走的可能性几乎为 0，所以他当机立断，判断罪犯绝无逃走的可能。

好了，故事说完了，好听吗？”梁老师笑着问。

大家也都笑了。

“其实这个故事要告诉我们的是，如何做一个善于分析思考的有心人。同样是警员，大多数人不注意观察，连茶都没有看到；少数警员观察到了，但是他看的除了茶水还是茶水，也不会去感受茶水的温度；而小余却能从茶杯到茶水，从茶水到水温多角度看待事物，难能可贵的是，他还能看从水温再联想到时间，这种细致入微与善于推理的能力最后让他获得了成功。

联想一下我之前总结的索引的三大特点，这三大特点我是如何引导大家推理出来的，一张索引结构图看似平淡无奇，但是我是从下到上，从外到里，从左到右三个不同的方向来引导大家，从而得出了三个重要的特点，这就是多角度细心看待事物的结果，这三个特点接下来将会有更加深远的意义，类似于小余从茶水温度判断罪犯逃离时间的这部分情节，精彩纷呈。希望大家后续能否像小余一样跟着我的思路进行思考学习。

接下来的过程，我估计大家会有一个很深刻的感触，就是，原来你就是小余！各位小余们，你们刚才说的那些 SQL，很快就可以比平时运行速度提高百倍甚至千倍都不在话下。”

5.2.1.5 妙用三特征之高度较低

1. 索引高度较低验证

“各位小余们……”

收获，不止 Oracle

梁老师故意放慢语气，台下笑成一片了。

“现在要开始好好利用我们总结的三大特点了，先从索引高度比较低开始说起，这里大家要注意，索引的大小和高度是有巨大区别的，可能大小差了好多倍，但是高度却一样，这点大家可以理解吧？”

“可以！”台下同学们笑着大声回答。

“好，我先证明一下我前面的推论是否正确，索引的高度确实是比较低吗？大家看我的一组试验，如下：

① 构造一系列表 T1 到 T7，记录数从 5 到 500 万依次以 10 倍的差额逐步增大。

```
drop table t1 purge;
drop table t2 purge;
drop table t3 purge;
drop table t4 purge;
drop table t5 purge;
drop table t6 purge;
drop table t7 purge;
SQL> create table t1 as select rownum as id ,rownum+1 as id2 from dual connect by level<=5;
表已创建。
SQL> create table t2 as select rownum as id ,rownum+1 as id2 from dual connect by level<=50;
表已创建。
SQL> create table t3 as select rownum as id ,rownum+1 as id2 from dual connect by level<=500;
表已创建。
SQL> create table t4 as select rownum as id ,rownum+1 as id2 from dual connect by level<=5000;
表已创建。
SQL> create table t5 as select rownum as id ,rownum+1 as id2 from dual connect by level<=50000;
表已创建。
SQL> create table t6 as select rownum as id ,rownum+1 as id2 from dual connect by level<=500000;
表已创建。
SQL> create table t7 as select rownum as id ,rownum+1 as id2 from dual connect by level<=5000000;
表已创建。
```

脚本 5-1 做索引高度较低应用试验前的构造表

② 分别对 ID 列建索引：

```
SQL> create index idx_id_t1 on t1(id);
索引已创建。
SQL> create index idx_id_t2 on t2(id);
索引已创建。
SQL> create index idx_id_t3 on t3(id);
索引已创建。
SQL> create index idx_id_t4 on t4(id);
```

```

索引已创建。
SQL> create index idx_id_t5 on t5(id);
索引已创建。
SQL> create index idx_id_t6 on t6(id);
索引已创建。
SQL> create index idx_id_t7 on t7(id);
索引已创建。

```

脚本 5-2 继续完成建索引的准备工作

察看这些索引的大小，差别很大，最小的只有 64KB，最大的有 98304KB。

```

SQL> select segment_name, bytes/1024
2   from user_segments
3   where segment_name in ('IDX_ID_T1', 'IDX_ID_T2', 'IDX_ID_T3', 'IDX_ID_T4',
4       'IDX_ID_T5', 'IDX_ID_T6', 'IDX_ID_T7');

```

SEGMENT_NAME	BYTES/1024

IDX_ID_T1	64
IDX_ID_T2	64
IDX_ID_T3	64
IDX_ID_T4	128
IDX_ID_T5	1024
IDX_ID_T6	9216
IDX_ID_T7	98304

已选择 7 行。

脚本 5-3 观察比较各个索引的大小

但是统计索引高度时，我们观察发现这些索引的高度相差无几，记录数最小是 5 条，最大是 500 万条，而高度最小是 BLEVEL=0 表示的 1 层，与高度最大是 BLEVEL=2 表示的 3 层，也就差了 2 层而已！

同学们这里要注意一下：其中 BLEVEL 表示高度，0 的取值第一个 BLOCK 还没被索引装满，还没产生管理的索引块，这个 0 可以理解为高度是 1 层，1 则表示高度为 2 层，以此类推。

```

SQL> select index_name,
2         blevel,
3         leaf_blocks,
4         num_rows,
5         distinct_keys,
6         clustering_factor
7   from user_ind_statistics

```

```
8      where table_name in( 'T1','T2','T3','T4','T5','T6','T7');
```

INDEX_NAME	BLEVEL	LEAF_BLOCKS	NUM_ROWS	DISTINCT_KEYS	CLUSTERING_FACTOR
-----	-----	-----	-----	-----	-----
IDX_ID_T1	0	1	5	5	1
IDX_ID_T2	0	1	50	50	1
IDX_ID_T3	1	2	500	500	1
IDX_ID_T4	1	11	5000	5000	9
IDX_ID_T5	1	110	50000	50000	101
IDX_ID_T6	2	1113	500000	500000	1035
IDX_ID_T7	2	11705	5000000	5000000	11462

脚本 5-4 观察比较各个索引的高度

这是梁老师精心构思准备的一个经典试验，大家仔细观察，都看明白了吗，有什么疑问吗？”

这组试验让同学们对索引高度有了一个直观的认识，虽然梁老师前面的推理已经让大家感受到第2层以上的索引块由于装的是指针，不容易被填满，但是现实中记录差距如此之大，而高度差别如此之小，还是让大家吃了一惊。

“梁老师，为什么 IDX_ID_T1、IDX_ID_T2、IDX_ID_T3 三个索引一样大啊，都是 64KB，他们的表记录可是 5、50 和 500 啊？”林君好像发现新大陆一样。

“哦，谁能回答林君的问题啊？”梁老师不回答转向问同学们。

“数据库的最小单位虽然是块，但是最小的空间分配单位却是区，一个段要存在至少需要含一个区，你说的三个索引，就是三个索引 SEGMENT。

比如一个区含有 8 个块，一个块有 8KB，段的最小空间就是 64KB 了，这是预分配的空间，即便是建一张空表，大小也是 64KB，在空表上建一个索引，这个索引大小还是 64KB，表和索引实际大小只要不超过 64KB，数据库查询都是 64KB 这么大，因为一次性已经分配给你一个区的大小了。但是一旦超过 64KB，即便超过一点点，分配到的空间可能马上就是 128KB 这么大了，对吧梁老师？”小莲一开始也有些发懵，但是看到这三个索引大小都是 64KB，猛然想明白了。

“林君同学，回头请小莲同学吃饭，感谢她帮你把前面的体系物理结构复习了一遍，这里请大家记得空间分配大小和实际使用大小是不同的。”梁老师笑着说。

“明白了，谢谢！梁老师，我还有疑问，为什么 7 张表记录数依次都有 10 倍的差别，怎么不少索引的 BLEVEL 是一样高呢？”

“还是请同学们来回答这个问题吧。”

“梁老师，我知道，BLEVEL=0 这个层面表示只有叶子块，无论 5 条还是 50 条，都不能把第 1 个索引块装满，无须填入第 2 个块，所以没有上层块来管理，所以都是 1 层高。

BLEVEL=1 这个层面表示这个阶段已经到第 2 层了，但是无论是 500 条还是 5000 条还是 50000 条，都不能让叶子块的数据量多到把第 2 层的第 1 个管理块填满，所以产生不到第 3 层，所以都

是2层高。

其他类似，梁老师，我说的对吗？”晶晶流利地回答。

“林君，你听懂了吗？”梁老师问。

“明白了，我想起前面您说过的了。”林君不好意思地笑了。

“今天你要多准备点钱了，还要再请晶晶一块去哦。”

台下笑成一片。

2. 索引高度较低妙用

“接下来我们来看看索引高度较低的这个特性能给我们带来什么激动人心的东西。”梁老师笑着说，“我们建了7张表，其中t6表有50万条记录，而t7表有500万条记录。大家说说这两条查询语句 `select * from t6 where id=10` 和 `select * from t7 where id=10`，它们都是返回一条记录，但是t7表记录是t6表的10倍，它们查询速度会不会差别很大？”

“会差别很大！”林君回答。

“差别不大！”小莲和晶晶同时回答。

“那谁快呢？”梁老师问到。

“当然是t7表的查询慢了！”同学们齐声回答，大家心里暗笑，这还要问吗，当我们是三岁小孩啊。

“梁老师也不知道你们回答得对不对，还是做试验来证明吧，先对t6表做查询，这里再次强调一下，梁老师每次试验执行效率时，所执行语句都会执行两遍以上，这是为了消除物理读和递归调用的干扰，保证公平公正。

```
SQL> set autotrace traceonly
```

```
SQL> set linesize 1000
```

```
SQL> set timing on
```

```
SQL> select * from t6 where id=10;
```

已用时间: 00: 00: 00.01

执行计划

Plan hash value: 1902844584

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	26	4 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	T6	1	26	4 (0)	00:00:01
* 2	INDEX RANGE SCAN	IDX_ID_T6	1		3 (0)	00:00:01

Predicate Information (identified by operation id):

收获，不止 Oracle

```
2 - access("ID"=10)
Note
-----
- dynamic sampling used for this statement
统计信息
-----
0 recursive calls
0 db block gets
5 consistent gets
0 physical reads
0 redo size
462 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 5-5 观察上述与 t6 表相关的索引扫描的性能

接下来执行 t7 语句的查询，如下：

```
SQL> select * from t7 where id=10;
已用时间: 00:00:00.01
执行计划
-----
Plan hash value: 1124755243

-----
| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time |
-----
| 0 | SELECT STATEMENT | | 1 | 26 | 4 (0) | 00:00:01 |
| 1 | TABLE ACCESS BY INDEX ROWID | T7 | 1 | 26 | 4 (0) | 00:00:01 |
|* 2 | INDEX RANGE SCAN | IDX_ID_T7 | 1 | | 3 (0) | 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
2 - access("ID"=10)
Note
-----
- dynamic sampling used for this statement
统计信息
-----
0 recursive calls
0 db block gets
```

```

5 consistent gets
0 physical reads
0 redo size
462 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 5-6 观察上述与 t7 表相关的索引扫描的性能

老师的试验做完了，我什么话都不说，同学们自己认真看结果，判断一下自己刚才的回答对不对。”

小莲有点看傻了眼了，返回的同样是 1 条，执行时间同样是 0.01 秒，产生的逻辑读同样是 5 个，计算的 COST 成本同样是 4，这说明两个语句执行得一样快！

“大家是不是有点不相信自己的眼睛，这两个试验表明虽然一张表是 50 万条记录，另一张表是 500 万条记录，但是利用索引来返回同样记录数的查询，效率居然分毫不差，大家知道原因吗？”梁老师问。

“梁老师，因为他们的高度 BLEVEL 都为 2，表示高度都为 3，所以产生的 IO 次数都一样，如果 500 万的记录恰好导致 BLEVEL 为 3，表示高度为 4 时，产生的 IO 次数就不会一样了，速度就会有差别了。”晶晶迅速起身回答。

“说得非常好，索引真的是一个利器，刚才是 500 万的记录，即便到 50 亿，这个树也不会太高的，顶多也就是到高度为 7，如果返回还是只有一条记录的话，那也就是比我们刚才的查询多了两三个 IO 而已，查询时间绝对能保证在 1 秒内完成的。”梁老师说。

经过这系列试验，大家对索引的印象非常深刻了。

“好了，现在我们看看，没有索引的情况下，查询需要花费多少时间，试验如下：

```
SQL> drop index IDX_ID_T6;
```

索引已删除。

```
已用时间: 00:00:00.06
```

```
SQL> select * from t6 where id=10;
```

执行计划

```
-----
Plan hash value: 1930642322
```

	Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
	0	SELECT STATEMENT		12	312	239 (3)	00:00:03
*	1	TABLE ACCESS FULL	T6	12	312	239 (3)	00:00:03

收获，不止 Oracle

Predicate Information (identified by operation id):

1 - filter("ID"=10)

Note

- dynamic sampling used for this statement

统计信息

0 recursive calls
0 db block gets
1041 consistent gets
0 physical reads
0 redo size
462 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

脚本 5-7 再次测试与 t6 表相关的全表扫描查询的性能

SQL> drop index IDX_ID_T7 ;

索引已删除。

SQL> select * from t7 where id=10;

执行计划

Plan hash value: 3979761704

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		128	3328	2595 (3)	00:00:32
* 1	TABLE ACCESS FULL	T7	128	3328	2595 (3)	00:00:32

Predicate Information (identified by operation id):

1 - filter("ID"=10)

Note

- dynamic sampling used for this statement

统计信息

```

0 recursive calls
0 db block gets
11474 consistent gets
0 physical reads
0 redo size
462 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 5-8 再次测试与 t7 表相关的全表扫描查询的性能

可以看出，由于删除了索引，针对 t6 表的查询时间为 0.03 秒，是原来的 3 倍，而针对 t7 表的查询为 0.32 秒，是原来的 30 倍。逻辑读更是差异明显，t6 表产生了 1041 次逻辑读，是原来的 200 倍，而 t7 表的查询产生了 11474 次逻辑读，是原来的 2000 倍！

由此得出结论，索引的这个高度不高的特性给查询带来了巨大的便捷，但是请注意我们的查询只返回 1 条记录，如果查询返回绝大部分的数据，那用索引反而要慢得多，这点大家知道为什么吗？”

“如果索引的高度为 3，查询到一条记录大致需要 3 到 4 次 IO，如果查询返回 100 万条记录，就是 100 万乘以 3 或 4，那就是三四百万的 IO 数量，还不如全表扫描吧，如果表有 100 万条记录，不可能需要三四百万个块来装 100 万个数据吧。梁老师，对吗？”小莲回答。

“说得非常好，大致就是这个道理，另外我还要补充一下，全表扫描还有一个优势，就是一次可以读取多个块，不仅是一次读取一个块，这样 IO 的次数还可以大大降下来的。”

小莲点点头，原来全表扫描还可以对块进行多块读，那刚才自己举的例子差距还可以更大了。

“最后再问问大家，有没有觉得奇怪，怎么全表扫描速度也这么快啊，500 万条记录怎么用 0.3 秒就完成了？”

“是啊，我早就想问了！”敬昱忍不住说出口！

“梁老师您举例的表的字段也太少了吧，才两个字段，一条记录占不了多少空间，所以记录数虽然有 500 万条，但是 Oracle 的块却不会占用很多，所以全表扫描逻辑读也不会太多，是这个原因吧。”小莲早就思考过这个问题了。

“小莲说得对，实际情况往往是，表记录有几十上百个字段，而不是仅仅两个，所以大家要意识到这一点，刚才如果老师把 t6 和 t7 表记录的字段同样类型长度地增加到 40 个，那块就会增加 20 倍，遍历的数据块当然也会增加 10 倍，速度也就慢下来了，大致要 6 秒了。

老师的优化工作中，曾经有一个经典的案例，就是通过业务分析，把一个有 80 多个字段的表的应用精简为 20 个字段，从而在需要使用全表扫描的场合下大大提升了性能。”

收获，不止 Oracle

梁老师的这系列试验有意构造出记录差异显著的表索引段大小可以相同、记录差异显著的表索引扫描性能可以相同、表字段越少查询性能越高等经典例子，让大家得以巩固所学知识，加深对索引高度的理解，确实是用心良苦，小莲感慨之余对梁老师又平添了几分尊重。

3. 分区索引设计误区

“大家还记得前面谈过的分区表吧，现在我们来谈谈分区表的索引，分区表的索引分为两种，一种是局部索引，一种是全局索引。局部索引等同于为每个分区段建分区的索引，从 `user_segment` 的数据字典中，我们可以观察到表有多少个分区，就有多少个分区索引的 `segment`，下面我做系列试验给大家看看。

先建分区表 `part_tab`，插入数据，并分别在 `col2` 上建局部索引，在 `col3` 上建全局索引。

```
SQL> drop table part_tab purge;
```

表已删除。

```
SQL> create table part_tab (id int,col2 int,col3 int)
```

```
2      partition by range (id)
```

```
3      (
```

```
4          partition p1 values less than (10000),
```

```
5          partition p2 values less than (20000),
```

```
6          partition p3 values less than (30000),
```

```
7          partition p4 values less than (40000),
```

```
8          partition p5 values less than (50000),
```

```
9          partition p6 values less than (60000),
```

```
10         partition p7 values less than (70000),
```

```
11         partition p8 values less than (80000),
```

```
12         partition p9 values less than (90000),
```

```
13         partition p10 values less than (100000),
```

```
14         partition p11 values less than (maxvalue)
```

```
15     )
```

```
16     ;
```

表已创建。

```
SQL> insert into part_tab select rownum,rownum+1,rownum+2 from dual connect by rownum <=110000;
```

已创建 110000 行。

```
SQL> commit;
```

提交完成。

```
SQL> create index idx_par_tab_col2 on part_tab(col2) local;
```

索引已创建。

```
SQL> create index idx_par_tab_col3 on part_tab(col3);
```

索引已创建。

脚本 5-9 分区索引相关试验的准备工作

随后我们观察可发现，分区表的每个分区就是一个段，这里显示有 11 个段，段的类型为 TABLE PARTITION。建了分区索引后，每个分区索引其实也就是一个段，这里显示也有 11 个段，每个段的类型为 INDEX PARTITION。而 col3 上建索引由于没有加 local 关键字，所以是全局索引，也就是普通索引，仅一个段，段的类型为 INDEX。

```
SQL> col segment_name format a20
```

```
SQL> select segment_name, partition_name, segment_type
```

```
2   from user_segments
```

```
3   where segment_name = 'PART_TAB';
```

SEGMENT_NAME	PARTITION_NAME	SEGMENT_TYPE

PART_TAB	P1	TABLE PARTITION
PART_TAB	P10	TABLE PARTITION
PART_TAB	P11	TABLE PARTITION
PART_TAB	P2	TABLE PARTITION
PART_TAB	P3	TABLE PARTITION
PART_TAB	P4	TABLE PARTITION
PART_TAB	P5	TABLE PARTITION
PART_TAB	P6	TABLE PARTITION
PART_TAB	P7	TABLE PARTITION
PART_TAB	P8	TABLE PARTITION
PART_TAB	P9	TABLE PARTITION

已选择 11 行。

```
SQL> select segment_name, partition_name, segment_type
```

```
2   from user_segments
```

```
3   where segment_name = 'IDX_PAR_TAB_COL2';
```

SEGMENT_NAME	PARTITION_NAME	SEGMENT_TYPE

IDX_PAR_TAB_COL2	P1	INDEX PARTITION
IDX_PAR_TAB_COL2	P10	INDEX PARTITION
IDX_PAR_TAB_COL2	P11	INDEX PARTITION
IDX_PAR_TAB_COL2	P2	INDEX PARTITION
IDX_PAR_TAB_COL2	P3	INDEX PARTITION
IDX_PAR_TAB_COL2	P4	INDEX PARTITION
IDX_PAR_TAB_COL2	P5	INDEX PARTITION
IDX_PAR_TAB_COL2	P6	INDEX PARTITION
IDX_PAR_TAB_COL2	P7	INDEX PARTITION
IDX_PAR_TAB_COL2	P8	INDEX PARTITION

```
IDX_PAR_TAB_COL2      P9                                INDEX PARTITION
```

已选择 11 行。

```
SQL> select segment_name, partition_name, segment_type
       2   from user_segments
       3   where segment_name = 'IDX_PAR_TAB_COL3';
```

SEGMENT_NAME	PARTITION_NAME	SEGMENT_TYPE
-----	-----	-----
IDX_PAR_TAB_COL3		INDEX

脚本 5-10 分区索引情况查看

以上分析是对之前知识的一个复习与补充，同学们应该都能明白。接下来，一个非常常见的设计误区出现了，请看下面系列试验。

建一张普通表 `norm_tab`，字段和记录数与 `part_tab` 一样，并且同样在 `col2` 和 `col3` 上分别建索引。

```
SQL> drop table norm_tab purge;
表已删除。
SQL> create table norm_tab (id int,col2 int,col3 int);
表已创建。
SQL> insert into norm_tab select rownum,rownum+1,rownum+2 from dual connect by rownum <=110000;
已创建 110000 行。
SQL> commit;
提交完成。
SQL> create index idx_nor_tab_col2 on norm_tab(col2);
索引已创建。
SQL> create index idx_nor_tab_col3 on norm_tab(col3);
索引已创建。
```

脚本 5-11 继续做准备工作，构建普通表及索引

我们来看看分别针对 `part_tab` 和 `norm_tab` 两表的 `col2` 列的查询效率如何，语句分别是 `select * from part_tab where col2=8` 和 `select * from norm_tab where col2=8`，依然采用 `set autotrace` 的跟踪方式，先查看针对 `part_tab` 的查询结果，发现索引扫描遍历了全部的 11 个分区，执行计划中我们可以观察到，**PSTART** 和 **PSTOP** 是从 1 到 11。

让该语句多次执行消除物理读和递归调用后，发现逻辑读为 24，COST 代价为 13，具体如下：

```
SQL> set autotrace traceonly
SQL> set linesize 1000
SQL> set timing on
```

```
SQL> select * from part_tab where col2=8 ;
```

```
已用时间: 00: 00: 00.01
```

```
执行计划
```

```
-----
```

```
Plan hash value: 2955748241
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT		1	14	13 (0)	00:00:01		
1	PARTITION RANGE ALL		1	14	13 (0)	00:00:01	1	11
2	TABLE ACCESS BY LOCAL INDEX ROWID	PART_TAB	1	14	13 (0)	00:00:01	1	11
* 3	INDEX RANGE SCAN	IDX_PAR_TAB_COL2	1		12 (0)	00:00:01	1	11

```
-----
```

```
Predicate Information (identified by operation id):
```

```
-----
```

```
3 - access("COL2">=8)
```

```
统计信息
```

```
-----
```

```
0 recursive calls
0 db block gets
24 consistent gets
0 physical reads
0 redo size
520 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 5-12 全分区索引扫描产生大量逻辑读

接下来查看 norm_tab 的查询结果，多次执行消除递归调用和物理读后，我们发现最终的逻辑读仅为 4，COST 代价仅为 2，如下：

```
SQL> select * from norm_tab where col2=8 ;
```

```
已用时间: 00: 00: 00.00
```

```
执行计划
```

```
-----
```

```
Plan hash value: 3649198428
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
----	-----------	------	------	-------	-------------	------

0	SELECT STATEMENT		1	14	2	(0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	NORM_TAB	1	14	2	(0)	00:00:01
* 2	INDEX RANGE SCAN	IDX_NOR_TAB_COL2	1		1	(0)	00:00:01

Predicate Information (identified by operation id):

2 - access("COL2">=8)

统计信息

0	recursive calls
0	db block gets
4	consistent gets
0	physical reads
0	redo size
520	bytes sent via SQL*Net to client
400	bytes received via SQL*Net from client
2	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
1	rows processed

脚本 5-13 普通表索引扫描逻辑读少得多

比较可以看出，针对分区表的查询逻辑读是针对普通表的逻辑读的 6 倍之多，COST 分别为 13 和 2，也有 6 倍之差距，这是为什么呢？”梁老师问。

“梁老师，通过前面展现的多个索引段可以看出来，分区表的索引等同于查询了 11 个小索引，而小索引虽然体积比整个大索引小很多，但是高度却相差无几，比如小索引高度为 3，大索引高度为 4，那遍历所有小索引的 IO 大致为 11 乘以 3 为 33，远大于 4，我发现您总结的这个规律真好用啊！”小莲不自觉地恭维了梁老师一句。

“说得非常好，我们来比较一下分区索引和普通索引的高度差别，发现此时大小虽然差别很大，但是高度居然一样，都是 BLEVEL=1，也就是 2 层的高度，换句话说，此时分区单个小索引的查询已经和全局大索引的查询的 IO 个数是一样的了，而分区索引的 IO 个数还要乘以分区个数，性能当然低下了。

```
SQL> select index_name,
       blevel,
       leaf_blocks,
       num_rows,
       distinct_keys,
       clustering_factor
FROM USER_IND_PARTITIONS
```

```
where index_name='IDX_PAR_TAB_COL2';
```

INDEX_NAME	BLEVEL	LEAF_BLOCKS	NUM_ROWS	DISTINCT_KEYS	CLUSTERING_FACTOR
IDX_PAR_TAB_COL2	1	21	9999	9999	24
IDX_PAR_TAB_COL2	1	23	10000	10000	28
IDX_PAR_TAB_COL2	1	23	10000	10000	28
IDX_PAR_TAB_COL2	1	23	10000	10000	28
IDX_PAR_TAB_COL2	1	23	10000	10000	28
IDX_PAR_TAB_COL2	1	23	10000	10000	28
IDX_PAR_TAB_COL2	1	23	10000	10000	28
IDX_PAR_TAB_COL2	1	23	10000	10000	28
IDX_PAR_TAB_COL2	1	23	10000	10000	28
IDX_PAR_TAB_COL2	1	23	10000	10000	28
IDX_PAR_TAB_COL2	1	23	10000	10000	28
IDX_PAR_TAB_COL2	1	23	10001	10001	28

已选择 11 行。

```
SQL> select index_name,
2         blevel,
3         leaf_blocks,
4         num_rows,
5         distinct_keys,
6         clustering_factor
7         from user_ind_statistics
8         where index_name ='IDX_NOR_TAB_COL2';
```

INDEX_NAME	BLEVEL	LEAF_BLOCKS	NUM_ROWS	DISTINCT_KEYS	CLUSTERING_FACTOR
IDX_NOR_TAB_COL2	1	244	110000	110000	299

脚本 5-14 分别观察分区表和普通表的索引高度

因此分区表索引的设计是有讲究的，如果设置了分区索引，但是却用不到分区条件，性能将继续下降，如果你建了分区索引，但是你又根本无法加上分区字段的条件，那建议你不要建分区索引。

假如刚才的语句允许加上分区字段的条件，比如增加 `id=2`，那结果就不一样了，`set autotrace traceonly` 跟踪加上多次反复执行后的最终结果如下：

```
SQL> select * from part_tab where col2=8 and id=7;
执行计划
```

Plan hash value: 702898905

Id	Operation	Name	Rows	Bytes	Cost	(%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT		1	11	2	(0)	00:00:01		
1	PARTITION RANGE SINGLE		1	11	2	(0)	00:00:01	1	1
* 2	TABLE ACCESS BY LOCAL INDEX ROWID	PART_TAB	1	11	2	(0)	00:00:01	1	1
* 3	INDEX RANGE SCAN	IDX_PAR_TAB_COL2	1		1	(0)	00:00:01	1	1

Predicate Information (identified by operation id):

2 - filter("ID"=7)

3 - access("COL2"=8)

统计信息

0 recursive calls

0 db block gets

4 consistent gets

0 physical reads

0 redo size

520 bytes sent via SQL*Net to client

400 bytes received via SQL*Net from client

2 SQL*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

1 rows processed

脚本 5-15 分区索引扫描仅落在某一分区，性能大幅提升

这下 **PSTART** 与 **PSTOP** 显示只遍历了 1 个分区，逻辑读也从 24 减少为 4 个，性能大幅度提升了。大家现在对分区表的索引的设计是否有了一个正确的认识了？”梁老师问。

大家纷纷点头。

“关于分区索引，也有过经典的案例，某表有近百个分区，但是查询中居然用不到分区条件，这样索引扫描时等同于扫描 100 个高度并没有低多少的小索引，结果导致系统运行得非常缓慢，后面梁老师将局部索引改为全局索引后，系统性能马上提升了近百倍。

还有一个是反过来的例子，某分区表有近千个分区，查询总能用到分区条件，而应用却未建分区索引，只建了全局索引。此时如果建分区索引，其大小将会是全局索引的千分之一，高度大致少了 2 层左右，也就是说每次查询的逻辑读会减少 2 次以上。这些查询一天可以执行上亿次，我的这个改造，让系统一天减少了几亿个逻辑读的产生，为缓解系统压力带来了巨大帮助。”梁老师继续说。

听了梁老师简单的案例分享，小莲忽然觉得有些震撼，真没想到，看似平淡无奇的“索引的高度比较低”这个推理总结，居然会对数据库的应用、甚至设计带来如此巨大的影响。

5.2.1.6 巧用三特征之存储列值

1. COUNT(*)优化

(1) 考虑用索引理论很完美

“各位小余们，接下来我们是要描述第二个特点了，就是索引存储列值及 rowid 的特性，这个特性听起来很普通，其实身边最常见的语句性能的提升却往往是基于对这个特点的认识。”梁老师接着说，“你们先分析看看最简单的统计条数的 SQL 语句和索引有啥关系，如何优化吧。

下面简单构造出 t 表，然后在 object_id 列建索引，并统计该表记录，具体如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t as select * from dba_objects;
表已创建。
SQL> create index idx1_object_id on t(object_id);
索引已创建。
SQL> select count(*) from t;
COUNT(*)
-----
55614
```

脚本 5-16 COUNT(*)优化试验前的建表及索引

这里我想问大家，请大家认真思考，这个简单的 `select count(*) from t` 到底能否用到索引，用索引是高效还是低效呢？”

“梁老师，统计表的记录数必须要遍历全表，我还是觉得用不到索引。”虽然前面的环节梁老师已经指明她不对了，但是小莲还是坚持自己的观点。

“梁老师，可以用到索引，并且用到索引一定高效！”

因为表的情况和索引情况的差别在于表是把整行的记录依次放进 BLOCK 形成 DATA BLOCK，而索引是把所在列的记录排序后依次放进 BLOCK 里形成 INDEX BLOCK，既然在没有索引的情况下，DATA BLOCK 中可以统计出表记录数，那 INDEX BLOCK 肯定也可以。方法就是前者汇总各个 DATA BLOCK 中的行的插入记录数，后者汇总各个 INDEX BLOCK 中的索引列的插入记录数。

最关键的是，INDEX BLOCK 里存放的值是表的特定的索引列，一列或者就几列，所需容纳空间要比存放整行也就是所有列的 DATA BLOCK 要少得多，假如当前这个 T 段的 DATA BLOCK 需要几百个块来容纳，或许索引段只需几十个块就够装了。所以用索引一定高效！

收获，不止 Oracle

梁老师，我说的对吧？”晶晶起身滔滔不绝，充满自信。

“说得非常好，梁老师有点佩服你了晶晶同学，讲述得有条有理、脉络分明且通俗易懂，你好好学，下期的新人培训可以由让你来上课了。”

梁老师的赞扬让晶晶有点不好意思，脸红了半边。小莲也有点不好意思，这么简单的道理，怎么没明白呢，虽然记得索引块能存储索引列值这个要点，但是没考虑到只要存进去就能数出来的简单生活道理，真是惭愧。

“小莲同学，你现在同意晶晶的观点吗，这个查询语句可以用到索引，并且一定高效，是吗？”

“是的，可以用到 `OBJECT_ID` 列上的索引！”小莲回答得很肯定。

“晶晶推理得那么完美，解释得那么通俗，梁老师也同意晶晶的观点了。让我们来看看试验的结果，看看事实是什么情况。

（2）未料到空值现实很残酷

前面表已经构造好了，索引也建立好了，我们依然用 `set autotrace on` 的方式来做跟踪，对了，梁老师多次用这种方法，希望大家工作中也能经常用到这个跟踪手段，接下来，激动人心的时刻开始了，看看执行计划中是否能用到索引。

```
SQL> set autotrace on
SQL> set linesize 1000
SQL> set timing on
SQL> select count(*) from t;
COUNT(*)
```

55614

已用时间: 00: 00: 00.08

执行计划

Plan hash value: 2966233522

Id	Operation	Name	Rows	Cost (%CPU)	Time
0	SELECT STATEMENT		1	175 (1)	00:00:05
1	SORT AGGREGATE		1		
2	TABLE ACCESS FULL	T	55614	175 (1)	00:00:05

统计信息

```
0 recursive calls
0 db block gets
770 consistent gets
0 physical reads
```

```

0 redo size
414 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 5-17 COUNT(*)在索引列有空值时无法用到索引

怎么样，用到索引了吗，梁老师问大家。”

台下沉默了一小会儿后，陆续有同学大声说没用到索引。

“不可能吧，TABLE ACCESS FULL 还真是全表扫描，太不好意思，梁老师演砸了！”梁老师故作惊奇。

“梁老师是不是演砸了？同学们都想想看，怎么会用不到索引呢？”梁老师问。

（3）空值有想到生活仍美好

同学们思考了很久没应答，小莲忽然想起来，索引不能存储空。她立即起身说：“梁老师，索引不能存储空记录，这样如果表的索引列有空的记录，那依据索引来统计表的记录数，肯定要出错啊，所以刚才用不到索引。”

“说得太好了，那我们现在改变一个写法，写成 `select count(*) from t where object_id is not null`，你们觉得这回可以用到索引吗？”梁老师问。

“应该可以。”受到上次试验结果的影响，同学的回答开始有些不确定了。

“好，让我们再做一次试验吧，看看是什么结果，如下：

```

SQL> set autotrace on
SQL> set linesize 1000
SQL> set timing on
SQL> select count(*) from t where object_id is not null;
COUNT(*)

```

55614

已用时间: 00: 00: 00.00

执行计划

Plan hash value: 1296839119

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	5	29 (0)	00:00:01

1	SORT AGGREGATE	1	5	
* 2	INDEX FAST FULL SCAN	IDX1_OBJECT_ID	55614	271K 29 (0) 00:00:01

Predicate Information (identified by operation id):

2 - filter("OBJECT_ID" IS NOT NULL)

统计信息

0	recursive calls
0	db block gets
130	consistent gets
0	physical reads
0	redo size
414	bytes sent via SQL*Net to client
400	bytes received via SQL*Net from client
2	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
1	rows processed

脚本 5-18 明确索引列非空，即可让 COUNT(*)用到索引

看来我告诉 Oracle 此列为非空后，终于用到索引了。执行计划从原先的 TABLE ACCESS FULL 变化为 INDEX FAST FULL SCAN，逻辑读由原先的 770 减少为 130，代价由原来的 175 减少为 29，性能大大提升了！

另外大家注意观察一下，其实该表的 OBJECT_ID 列并没有空值，因为加上 where object_id is not null 后取值和未加的一样，都是 55614 条，不过你没告诉 Oracle 此列不允许为空，Oracle 是不会冒这个险的。

（4）多条道路通往幸福彼岸

同学们，你们现在动动脑子，如果我不写 where object_id is not null，语句依然要保持和 select count(*) from t 一样，怎么用到索引？”梁老师开始考大家。

“梁老师，把这列的属性改为非空！”梁老师话音刚落晶晶就抢答了，小姑娘真是思维敏捷。

“说得太好了，我们来做个试验看看，由于表是 CREATE 命令建立起来的，所以当前列的属性全部允许为空，如下：

SQL> desc t;

Name	Type	Nullable
OWNER	VARCHAR2(30)	Y
OBJECT_NAME	VARCHAR2(128)	Y

SUBOBJECT_NAME	VARCHAR2(30)	Y
OBJECT_ID	NUMBER	Y
DATA_OBJECT_ID	NUMBER	Y
OBJECT_TYPE	VARCHAR2(19)	Y
CREATED	DATE	Y
LAST_DDL_TIME	DATE	Y
TIMESTAMP	VARCHAR2(19)	Y
STATUS	VARCHAR2(7)	Y
TEMPORARY	VARCHAR2(1)	Y
GENERATED	VARCHAR2(1)	Y
SECONDARY	VARCHAR2(1)	Y

脚本 5-19 查看 T 表的列是否为空

更改 OBJECT_ID 列的属性，修改为不允许为空，如下：

```
SQL> alter table t modify OBJECT_ID not null;
表已更改。
```

脚本 5-20 修改 object_id 列为非空

接下来，我们可以开始测试这个 COUNT(*) 语句能否用到索引了：

```
SQL> set autotrace on
SQL> set linesize 1000
SQL> set timing on
SQL> select count(*) from t ;
COUNT(*)
```

55614

已用时间: 00: 00: 00.00

执行计划

Plan hash value: 1296839119

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	5	29 (0)	00:00:01
1	SORT AGGREGATE		1	5		
* 2	INDEX FAST FULL SCAN	IDX1_OBJECT_ID	55614	271K	29 (0)	00:00:01

Predicate Information (identified by operation id):

2 - filter("OBJECT_ID" IS NOT NULL)

统计信息	

0	recursive calls
0	db block gets
130	consistent gets
0	physical reads
0	redo size
414	bytes sent via SQL*Net to client
400	bytes received via SQL*Net from client
2	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
1	rows processed

脚本 5-21 这下不改 SQL，让 COUNT(*)用到索引

这下我们发现，只要指定了索引列的属性为非空，查询语句就高效地利用到了索引，和之前的 select count(*) from t where object_id is not null 语句一样高效。

好了，除了 is not null 指定可以将列的属性修改为不允许空外，还有其他方法吗？”梁老师继续提问。

“我想到了，主键本身不允许为空，所以如果 OBJECT_ID 列是主键就可以了！”小莲说。

“非常好，我们继续试验验证一下。

SQL> drop table t purge;					
表已删除。					
SQL> create table t as select * from dba_objects;					
表已创建。					
SQL> alter table t add constraint pk1_object_id primary key (OBJECT_ID);					
表已更改。					
SQL> set autotrace on					
SQL> set linesize 1000					
SQL> set timing on					
SQL> select count(*) from t;					
COUNT(*)					

55614					
已用时间: 00: 00: 00.01					
执行计划					

Plan hash value: 1604907147					

Id	Operation	Name	Rows	Cost (%CPU)	Time

	0	SELECT STATEMENT				1		29	(0)	00:00:01	
	1	SORT AGGREGATE				1					
	2	INDEX FAST FULL SCAN		PK1_OBJECT_ID		64859		29	(0)	00:00:01	

Note

- dynamic sampling used for this statement

统计信息

0	recursive calls
0	db block gets
122	consistent gets
0	physical reads
0	redo size
414	bytes sent via SQL*Net to client
400	bytes received via SQL*Net from client
2	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
1	rows processed

脚本 5-22 object_id 列为主键，也可说明了非空属性

执行计划为 INDEX FAST FULL SCAN，果然用到了索引，代价和逻辑读比全表扫描少得多！怎么样同学们，都听明白了吗？”

小莲心中有些激动，自己经常写统计表记录的语句，但是在梁老师上课之前，她根本不知道这个如此简单的语句还能有优化的空间，而且原理又是如此浅显易懂。

（5）成功源于多角度的思考

“梁老师再问大家一下，什么时候在表的非空列建有索引时，COUNT(*)语句用索引效率不如全表扫描？”

同学们想了半天，没人回答。

“没人想到啊？”梁老师笑了，“如果一张表仅有一个字段，这个索引不是比表还大（多了rowid），那从索引中回答问题，效率还高吗？”

同学们都笑了。

“那什么时候 COUNT(*)查询语句用索引扫描比全表扫描高效很多呢？”梁老师继续提问。

“我知道了，表的字段很多，并且字段长度大多都很长，其中有一个非空且长度很短的列建了一个索引，这时索引的体积相对表来说特别小，那索引读效率就高多了。”小莲这回反应非常快了。

收获，不止 Oracle

梁老师鼓掌表示赞同。

2. SUM/AVG 优化

(1) 理想再次被空值击碎

“好了，接下来我们说说同样常见的 sum()和 avg 之类的聚合语句，具体语句如 select sum(object_id) from t 等，我们先构造例子如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t as select * from dba_objects;
表已创建。
SQL> create index idx1_object_id on t(object_id);
索引已创建。
```

脚本 5-23 SUM/AVG 优化试验准备之表及索引构建

现在我们即将执行语句 select sum(object_id) from t 如下，大家觉得能否用到索引？”

“可以！”台下都回答得很干脆。

“好，我们试验看看，到底能否用到索引。

```
SQL> set autotrace on
SQL> set linesize 1000
SQL> set timing on
SQL> select sum(object_id) from t;
SUM(OBJECT_ID)
-----
1929332122
已用时间: 00: 00: 00.03
执行计划
-----
Plan hash value: 2966233522

-----
| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time |
-----
| 0 | SELECT STATEMENT | | 1 | 13 | 175 (1) | 00:00:03 |
| 1 | SORT AGGREGATE | | 1 | 13 | | |
| 2 | TABLE ACCESS FULL | T | 64859 | 823K | 175 (1) | 00:00:03 |
-----

Note
-----
- dynamic sampling used for this statement
统计信息
```

```

-----
0 recursive calls
0 db block gets
770 consistent gets
0 physical reads
0 redo size
422 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 5-24 SUM/AVG 用不到索引（因为列允许为空）

有用到索引吗？”梁老师笑着问大家。

“没道理啊，索引不是已经存储了 OBJECT_ID 的列值，足够做 SUM 的运算啊！”敬昱纳闷地说。

（2）周全考虑迈出新天地

“我知道了，是因为这个列必须指定为非空。”晶晶说。

“说得很好，此时的情况和 COUNT(*)的情况类似，指定为非空或者是列属性为非空，即可走索引，如下：

```

SQL> set autotrace on
SQL> set linesize 1000
SQL> select sum(object_id) from t where object_id is not null;
SUM(OBJECT_ID)

```

```

-----
1929332122

```

执行计划

```

-----
Plan hash value: 1296839119

```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	13	31 (0)	00:00:01
1	SORT AGGREGATE		1	13		
* 2	INDEX FAST FULL SCAN	IDX1_OBJECT_ID	64859	823K	31 (0)	00:00:01

```

-----
Predicate Information (identified by operation id):

```

```

-----
2 - filter("OBJECT_ID" IS NOT NULL)

```

收获，不止 Oracle

```
Note
-----
- dynamic sampling used for this statement
统计信息
-----
0 recursive calls
0 db block gets
130 consistent gets
0 physical reads
0 redo size
422 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 5-25 在说明索引列非空后，SUM/AVG 可用到索引

此时果然用到索引，而且逻辑读才 130，相比用不到索引时的 770，性能可谓有大幅度的提升，AVG 语句与 SUM 语句类似，注意空值情况即可。这里就不多试验了，下面试验一个 SUM 和 AVG 及 COUNT 写在一起的情况，如下写法堪称经典，仅一次索引扫描即可完成，请大家好好体会：

```
SQL> select sum(object_id) ,avg(object_id),count(*) from t where object_id is not null;
SUM(OBJECT_ID) AVG(OBJECT_ID) COUNT(*)
-----
1929332122      34691.4828      55614
执行计划
-----
Plan hash value: 1296839119

| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time |
-----|-----|-----|-----|-----|-----|-----|
| 0 | SELECT STATEMENT | | 1 | 13 | 31 (0) | 00:00:01 |
| 1 | SORT AGGREGATE | | 1 | 13 | | |
|* 2 | INDEX FAST FULL SCAN | IDX1_OBJECT_ID | 64859 | 823K | 31 (0) | 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
2 - filter("OBJECT_ID" IS NOT NULL)
Note
-----
```

- dynamic sampling used for this statement
统计信息

```

0 recursive calls
0 db block gets
130 consistent gets
0 physical reads
0 redo size
571 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 5-26 SUM、AVG、COUNT 综合写法试验

大家注意到没，这个 COUNT(*)、SUM、AVG 连续三个聚合语句写在一起，逻辑读和单个 COUNT(*)、SUM 运算时性能一样，都是 130 个逻辑读。就是因为一次扫描索引块，可以同时解决三个问题，所以效率一样高。

不过列有空值理应不影响在索引中进行 SUM 和 AVG 等运算的，这里未指明非空则无法用到索引，其实是不应该的，这是优化器的缺陷。好在 select sum(object_id) from t where object_id is not null 和 select sum(object_id) from t 两语句等价，请大家在编写 SQL 时多留意。”

“原来身边这么多熟悉的 SQL 可以利用索引来优化，我天天写 COUNT 统计记录，都不知道这其中的奥秘。”小莲想着又平添了几分感慨。

3. MAX/MIN 优化

(1) 空值终于无法阻碍理想

“说了这么多熟悉的语句，接下来我们再查看一个使用非常频繁的取最大最小值的 SQL 语句，即 MAX 和 MIN 的研究，首先构造结构如下：

```

SQL> drop table t purge;
表已删除。
SQL> create table t as select * from dba_objects;
表已创建。
SQL> create index idx1_object_id on t(object_id);
索引已创建。

```

脚本 5-27 MAX/MIN 试验前的准备工作

现在我们来看看，语句 select max(object_id) from t 能用到索引吗，效率高吗？”梁老师问。

收获，不止 Oracle

由于每次都猜错，台下的学生们都有些紧张了。

“用不到索引，因为 object_id 允许为空，没写上 is not null。”小莲回答后心想，这梁老师还真鬼，每次都设陷阱让我们答错，这回总算提防住了。

“同学们的意见呢？”梁老师问。

“同意小莲的看法，不能用到索引。”同学们都支持小莲。

“好，那我们用试验来说话吧，现在这列的属性肯定是允许为空的，我也不加 is not null，执行如下：

```
SQL> select max(object_id) from t;
MAX(OBJECT_ID)
-----
      121477
执行计划
-----
Plan hash value: 692082706

-----
| Id | Operation                      | Name                | Rows  | Bytes | Cost (%CPU)  | Time      |
-----
|  0 | SELECT STATEMENT                |                     |      1 |    13 |       175 (1) | 00:00:03 |
|  1 |   SORT AGGREGATE                |                     |      1 |    13 |                |           |
|  2 |    INDEX FULL SCAN (MIN/MAX)    | IDX1_OBJECT_ID | 53391 | 677K |                |           |
-----

Note
-----
- dynamic sampling used for this statement
统计信息
-----
      0 recursive calls
      0 db block gets
      2 consistent gets
      0 physical reads
      0 redo size
    420 bytes sent via SQL*Net to client
    400 bytes received via SQL*Net from client
      2 SQL*Net roundtrips to/from client
      0 sorts (memory)
      0 sorts (disk)
      1 rows processed
```

脚本 5-28 MAX/MIN 语句应用索引非常高效

现在什么情况呢同学们，能用索引吗，你们猜对了吗？”梁老师笑着问。

被梁老师忽悠得三次都猜错了，同学们都笑了，不过印象却是够深刻了。

“其实我在前面 SUM 和 AVG 统计时就说过了，这个列的属性是否为空不应该影响能否使用索引作为‘瘦表’查询，但是奇怪的是 MAX/MIN 时无论列是否为空都可以用到索引，而 SUM/AVG 等聚合查询却必须要列为空方可用到索引。大家记住就好了，其实此类语句在运算时有无加上 is not null 的取值都是等价查询的，而 COUNT(*)则不一样，有无 is not null 的取值可是不等价的！”

(2) MAX/MIN 的惊天力量

空值问题我们暂且讨论到这，现在大家观察一下上述试验，觉得有没有什么特别之处？”

“我发现了，您从做索引相关试验开始，出现过三次有差别索引的相关执行计划，现在这个是 INDEX FULL SCAN (MIN/MAX)，在讲述如何利用索引高度比较低的特性时出现过 INDEX RANGE SCAN，在讲述 COUNT(*)和 SUM 等语句的优化时出现过 INDEX FAST FULL SCAN，怎么索引有这么多种扫描类型，它们区别在哪啊？”晶晶认真观察后，起身回答。

“真是个心细的女孩，不错！”梁老师表扬了晶晶，不过暂时没回答她的问题，“同学们，还有其他发现吗？”

“我发现了，逻辑读才 2！”小莲惊叫起来，“前面的各类查询逻辑读都没有这么低过，至少都在 100 以上啊。”

“说得非常好，COUNT(*)和 SUM/AVG 等都在 130 个逻辑读左右，但是这些语句有一个共同点，即都是属于吞吐量层面的操作，怎么会差别这么大呢，差了 60 倍还不止，不可思议吧。”梁老师笑着说。

同学们认真地看了看，确实如此，逻辑读居然才 2，如此之少，真有些难以想象。

“晶晶问了一些关于索引执行计划的差别，我先解释 INDEX FULL SCAN (MIN/MAX)吧，解释这个之前我们再回忆一下索引的结构图，还记得我让大家从左向右看时的演示图吗？想不起来再看看图 5-12。

这个 INDEX FULL SCAN (MIN/MAX)只需要 2 个逻辑读就完成查询的秘密在于，MAX 取值只需要往最右边的叶子块去瞧一瞧就行了，MAX 的取值一定在最右边的块上，块里的最后一行就是。而 MIN 取值，仅往最左边的块里去望一望即可了，最小值一定在里头，块里的第一行记录就是。

现在大家应该明白这个 INDEX FULL SCAN (MIN/MAX)的奥妙和优势了？这是一个很经典的思路，其实既包含了索引可以存储空值的技巧，又结合了索引是有序的技巧，堪称经典。”

同学们都纷纷点头，他们感觉很有收获。

“现在我继续问问大家，现在 t 表才 5 万多条记录，如果变成 100 万甚至更大，查询速度会影响很大吗？”

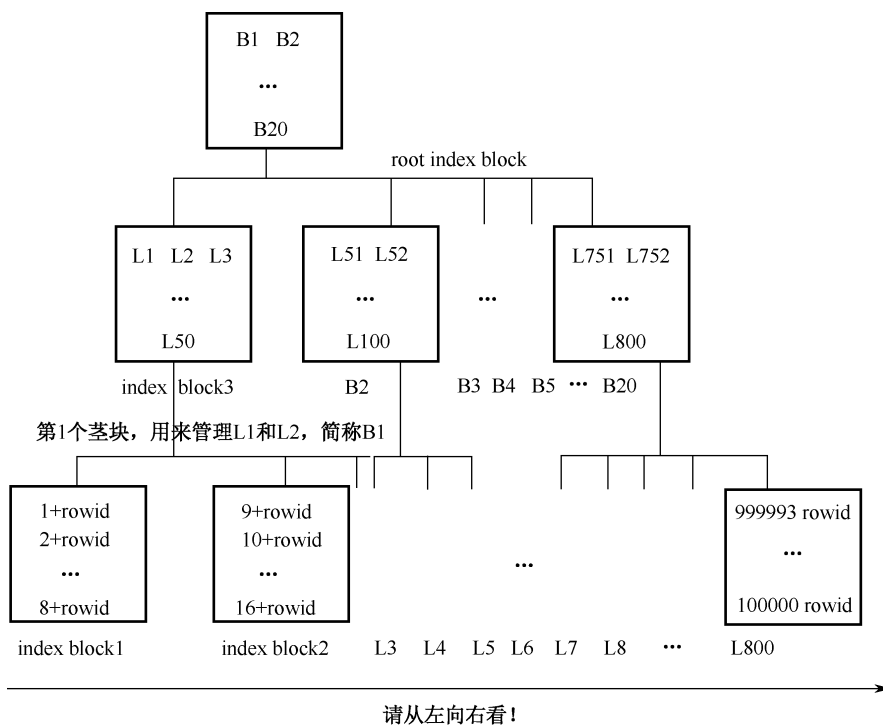


图 5-12 再次从左向右看

“不会，因为不管索引变成多大，始终都是最大值取最右边的叶子索引块，最小值取最左边的叶子索引块！”晶晶坚定地回答。

“如果晶晶说的是对的，那对于海量记录的表，只要是取最大或最小值，我们就不担心性能了，有这么好的事吗？我们做试验如下。

首先构造 **t_max** 表记录达到近 200 万条，然后建上索引，方法如下：

```
SQL> create table t_max as select * from dba_objects;
表已创建。
SQL> create index idx_t_max_obj on t_max(object_id);
索引已创建。
SQL> insert into t_max select * from t_max;
已创建 55615 行。
SQL> insert into t_max select * from t_max;
已创建 111230 行。
SQL> insert into t_max select * from t_max;
已创建 222460 行。
SQL> insert into t_max select * from t_max;
已创建 444920 行。
```

```
SQL> insert into t_max select * from t_max;
已创建 889840 行。
SQL> commit;
提交完成。
SQL> select count(*) from t_max;
COUNT(*)
-----
1779680
```

脚本 5-29 测 MAX 性能前的准备，构建一张大表

然后开始观察产生的逻辑读个数，这里我把 MIN 语句也统计了一下，免得大家认为 MIN 情况不一样，具体如下：

```
SQL> set autotrace on
SQL> set linesize 1000
SQL> select max(object_id) from t_max;
MAX(OBJECT_ID)
-----
121479
已用时间: 00: 00: 00.03
执行计划
-----
Plan hash value: 1235166074
-----
| Id | Operation                      | Name                | Rows  | Bytes | Cost (%CPU) | Time      |
-----
| 0 | SELECT STATEMENT                |                     | 1     | 13    | 10259 (1)   | 00:02:04 |
| 1 | SORT AGGREGATE                  |                     | 1     | 13    |              |          |
| 2 | INDEX FULL SCAN (MIN/MAX) | IDX_T_MAX_OBJ | 2231K | 27M   |              |          |
-----
Note
-----
- dynamic sampling used for this statement
统计信息
-----
0 recursive calls
0 db block gets
3 consistent gets
0 physical reads
0 redo size
420 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
```

收获，不止 Oracle

```

      2  SQL*Net roundtrips to/from client
      0  sorts (memory)
      0  sorts (disk)
1 rows processed

SQL> select min(object_id) from t_max;
MIN(OBJECT_ID)
-----
          2
已用时间: 00: 00: 00.03
执行计划
-----
Plan hash value: 1235166074

-----
| Id | Operation                      | Name           | Rows  | Bytes | Cost (%CPU)| Time      |
-----
|  0 | SELECT STATEMENT                |                |      1 |    13 | 10259  (1)| 00:02:04 |
|  1 |  SORT AGGREGATE                 |                |      1 |    13 |           |          |
|  2 |   INDEX FULL SCAN (MIN/MAX)     | IDX_T_MAX_OBJ | 2231K |   27M |           |          |
-----

Note
-----
- dynamic sampling used for this statement
统计信息
-----
      0  recursive calls
      0  db block gets
      3  consistent gets
      0  physical reads
      0  redo size
    418  bytes sent via SQL*Net to client
    400  bytes received via SQL*Net from client
      2  SQL*Net roundtrips to/from client
      0  sorts (memory)
      0  sorts (disk)
      1  rows processed
```

脚本 5-30 表大小差异明显，MAX/MIN 的性能却几无差异

记录数从 5 万到近 200 万后，逻辑读仅从 2 变为 3，而从查询的时间来看，MAX 和 MIN 的查询速度都是只花费了 0.03 秒就完成了，如果这个表没有索引，那查询速度将慢成千上万倍！

这就是这个 INDEX FULL SCAN (MIN/MAX)算法真正神奇的地方了，无论记录如何增大，

查询都不会太慢，我曾接触过数据达到近百亿的表，MAX 该表索引列的查询也不过在 1 秒不到的时间内就完成了，如果这个列没有索引，这个统计的查询将是一场噩梦！”

“梁老师，我真想回去试验一下，我是做数据仓库的，每次都对超级大表做 MAX 和 MIN 的统计，但是我们都没有对统计列建索引，因为同事说超级大表做聚合统计时索引没用，现在我好想马上回去试验一下。”快嘴的敬昱居然兴奋地打断了梁老师的话。

（3）MAX/MIN 的性能陷阱

“不错，有什么心得到时和大家分享一下。”梁老师笑着说，“接下来，关于 MAX 和 MIN 的统计，我还要再问大家一个问题，select min(object_id),max(object_id) from t 的写法能否用到索引，如果能用到索引，是否是 INDEX FULL SCAN (MIN/MAX)的扫描方式？”

“一样吧，能用到索引，应该也是 INDEX FULL SCAN (MIN/MAX)的扫描方式。”小莲犹豫了一会儿才回答，不过她的心里有些忐忑不安，梁老师每次的提问大家都回答错了，这次担心老师又在设套了。

“其他同学没回答估计是和小莲一个答案了，那我们来做一个试验，如下：

```
SQL> set autotrace on
SQL> set linesize 1000
SQL> select min(object_id),max(object_id) from t ;
MIN(OBJECT_ID) MAX(OBJECT_ID)
```

```
-----
                2          121477
已用时间: 00: 00: 00.03
执行计划
```

```
-----
Plan hash value: 2966233522
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	13	175 (1)	00:00:03
1	SORT AGGREGATE		1	13		
2	TABLE ACCESS FULL	T	53391	677K	175 (1)	00:00:03

```
-----
Note
```

```
-----
- dynamic sampling used for this statement
```

```
统计信息
```

```
-----
0 recursive calls
0 db block gets
770 consistent gets
```

收获，不止 Oracle

0	physical reads
0	redo size
487	bytes sent via SQL*Net to client
400	bytes received via SQL*Net from client
2	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
1	rows processed

脚本 5-31 MIN 和 MAX 同时写的优化（空值导致用不到索引）

有没有傻眼了，变成全表扫描了，难道又是空值的问题，我们增加 is not null 看看，发现走索引了，情况如下：

```
SQL> set autotrace on
SQL> set linesize 1000
SQL> select min(object_id),max(object_id) from t  where object_id is not null;
MIN(OBJECT_ID) MAX(OBJECT_ID)
-----
                2          121477
已用时间: 00: 00: 00.09
执行计划
-----
Plan hash value: 1296839119
-----
| Id | Operation                | Name                | Rows | Bytes | Cost (%CPU) | Time      |
-----
|  0 | SELECT STATEMENT          |                     |    1 |    13 |       31 (0) | 00:00:01 |
|  1 |   SORT AGGREGATE          |                     |    1 |    13 |              |           |
|*  2 |    INDEX FAST FULL SCAN    | IDX1_OBJECT_ID      | 53391 | 677K  |       31 (0) | 00:00:01 |
-----
Predicate Information (identified by operation id):
-----
   2 - filter("OBJECT_ID" IS NOT NULL)
Note
-----
   - dynamic sampling used for this statement
统计信息
-----
                0  recursive calls
                0  db block gets
            130  consistent gets
                0  physical reads
```

```

0 redo size
487 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 5-32 MIN 和 MAX 同时写的优化（无法用 INDEX FULL SCAN (MIN/MAX)）

不过虽然走索引了，但是扫描方式却是 INDEX FAST FULL SCAN 而非 INDEX FULL SCAN (MIN/MAX)，真是折磨死人了，不过逻辑读对比全表扫描，从 770 减少到 130，还是改进了不少。

到底是怎么回事呢？原来面对这种 `min(object_id),max(object_id)` 写法，Oracle 无法用 INDEX FULL SCAN (MIN/MAX) 这个算法同时在最左边和最右边读取，大家想想有什么好办法吗？”

“梁老师，我知道了，简单，其实把 MAX 和 MIN 分成两条语句统计不就行了？何必要连在一起呢？分成两条语句，逻辑读是 $2+2=4$ ，比这个 130 少多了吧。”晶晶忽然想到这点。

“其实晶晶在这里犯了一个大错误，谁知道是什么？”梁老师问。

“不好意思，我知道了，分开两条语句写，最大值和最小值并不对应，MAX 和 MIN 语句是先后执行的，数据库的记录已经变了，而 `min(object_id),max(object_id)` 才能保证是同时执行的。”晶晶思维真敏捷，这么快就意识到了自己的错误了。

“不错，反应真快！”梁老师表扬了晶晶，然后继续说，“大部分情况下，在实现同等功能的前提下，语句写得越复杂越慢，越简单越快，这次梁老师要做一个例外的情况，请大家看我的如下试验中的语句和性能：

```

SQL> set autotrace on
SQL> set linesize 1000
SQL> set timing on
SQL> select max, min
         from (select max(object_id) max from t ) a, (select min(object_id) min from t) b;

```

MAX MIN

121477 2

已用时间: 00: 00: 00.02

执行计划

Plan hash value: 251798682

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	26	350 (1)	00:00:05

收获，不止 Oracle

	1		NESTED LOOPS				1		26		350	(1)		00:00:05	
	2		VIEW				1		13		175	(1)		00:00:03	
	3		SORT AGGREGATE				1		13						
	4		INDEX FULL SCAN (MIN/MAX)		IDX1_OBJECT_ID		53391		677K						
	5		VIEW				1		13		175	(1)		00:00:03	
	6		SORT AGGREGATE				1		13						
	7		INDEX FULL SCAN (MIN/MAX)		IDX1_OBJECT_ID		53391		677K						

Note

- dynamic sampling used for this statement

统计信息

0 recursive calls
0 db block gets
4 consistent gets
0 physical reads
0 redo size
465 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

注：另一写法供参考，效率是一样的

```
SELECT (select max(object_id) max from t) max_id  
      , (select min(object_id) min from t) min_id  
FROM DUAL;
```

脚本 5-33 有趣的改写，完成了 MAX/MIN 同时写的最佳优化

总共有 4 个逻辑读，梁老师欣喜地发现，原来 2+2 真的等于 4 啊！”

同学们呆了半晌终于明白了梁老师的意思了，之前单独执行 MAX 和 MIN 都是 2 个逻辑读，而现在这个合并语句的逻辑读是 4。大家乐得都笑了。

“在老师优化过的案例中，就写过这个类似的 SQL 语句优化了数据库的性能，目前还跑在生产系统中，大家好好揣摩一下，这个写法是等价的，我为什么会写出这样的语句，那是因为我比较深刻地理解了索引的结构图，很清楚 INDEX FULL SCAN (MIN/MAX)这个索引相关执行计划的原理，最后不断尝试得出的优化思路，希望大家好好体会。

到现在为止，所有的优化相关的思路，都源于一个简单的索引结构图的不断推理分析，简单的背后不简单，好比小余看茶水捉住罪犯一样，不断思考和深入分析、推理，是解决问题的根本。

大家休息一下吧。”梁老师喝了一大口水，看来说得有些口舌干燥了。

同学们都安安静静地坐着，没见人起身离开。不少人没有回过神来，梁老师的课程让大家感触太深了。

4. 索引回表与优化

(1) 你确定查询需要返回所有字段吗

“看来大家都不想休息啊，这么有激情啊，太让人感动了，那老师就继续吧。”

接下来老师将会讲述执行计划中与索引相关的一个非常重要的知识点，索引回表读（TABLE ACCESS BY INDEX ROWID），先做一个简单的试验如下：

```
SQL> drop table t purge;
```

表已删除。

```
SQL> create table t as select * from dba_objects;
```

表已创建。

```
SQL> create index idx1_object_id on t(object_id);
```

索引已创建。

```
SQL> set autotrace traceonly
```

```
SQL> set linesize 1000
```

```
SQL> set timing on
```

```
SQL> select * from t where object_id<=5;
```

已用时间: 00:00:00.01

执行计划

Plan hash value: 2486998213

	Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
	0	SELECT STATEMENT		4	708	3 (0)	00:00:01
	1	TABLE ACCESS BY INDEX ROWID	T	4	708	3 (0)	00:00:01
*	2	INDEX RANGE SCAN	IDX1_OBJECT_ID	4		2 (0)	00:00:01

Predicate Information (identified by operation id):

2 - access("OBJECT_ID"<=5)

Note

- dynamic sampling used for this statement

统计信息

0 recursive calls
0 db block gets
5 consistent gets

0	physical reads
0	redo size
1310	bytes sent via SQL*Net to client
400	bytes received via SQL*Net from client
2	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
4	rows processed

脚本 5-34 索引回表读（TABLE ACCESS BY INDEX ROWID）的例子

这里有一个很显然的问题就是，从索引中可以读到索引列的信息，但是不可能读到该列以外的其他列的信息，这里的查询是 `select * from t where object_id<=5`，这个*表示该表所有字段都需要返回，因此必然是在扫描索引块中定位到具体 `object_id<=5` 这部分索引块后，再根据这部分索引块的 `rowid` 定位到 `t` 表所在的数据块，然后从数据块中获取到其他字段的记录，这就是上述例子中执行计划里 **TABLE ACCESS BY INDEX ROWID** 的含义。

我处理过非常多的类似案例，就是明明只需要返回少数字段，但是开发人员为了写 SQL 方便而写了*，这种情况在开发中非常常见，比如刚才的语句，可能只需查询返回 `object_id` 的列，但是写成了*。

如果真的 `select * from t where object_id<=5` 可以修正为 `select object_id from t where object_id<=5` 的话，那代码的效率将会有很大的差别，具体试验如下：

```
SQL> set autotrace traceonly
SQL> set linesize 1000
SQL> set timing on
SQL> select object_id from t where object_id<=5;
已用时间: 00: 00: 00.00
执行计划
-----
Plan hash value: 1056850546
-----
| Id | Operation          | Name           | Rows  | Bytes | Cost (%CPU) | Time      |
-----
| 0 | SELECT STATEMENT    |                |      4 |    52 |      2 (0) | 00:00:01 |
|* 1 |  INDEX RANGE SCAN | IDX1_OBJECT_ID |      4 |    52 |      2 (0) | 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
   1 - access("OBJECT_ID"<=5)
Note
-----
```

```
- dynamic sampling used for this statement
统计信息
```

```
-----
0 recursive calls
0 db block gets
3 consistent gets
0 physical reads
0 redo size
468 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
4 rows processed
```

脚本 5-35 比较消除 TABLE ACCESS BY INDEX ROWID 的性能

发现逻辑读从 5 减少为 3，代价从 3 缩减为 2，大家知道主要是因为什么吗？”梁老师问。

“因为 `select object_id from t where object_id <= 5` 的查询中索引就可以提供 `object_id` 列的返回信息了，少了一个 TABLE ACCESS BY INDEX ROWID 的动作，所以性能就能提升。”晶晶立即做出了回答。

“很好，大家不要小看这个了，我经历的优化案例中，消除 TABLE ACCESS BY INDEX ROWID 这个回表动作改进性能的案例占比达 20% 左右，相当常见。有时就是根据业务需求，把多余的字段取消了，恰好留下索引字段，就避免了回表。有时是某些字段不能取消展现，考虑联合索引的方式来避免回表，关于后者，等一下再描述。

对于梁老师的这个试验证明，大家有什么疑问吗？”

“没疑问！”从同学们这么干脆的回答来看，大家觉得这个都好理解。

(2) 索引不含查询列可考虑组合索引

“其实这里头还是很有玄机的，关于 TABLE ACCESS BY INDEX ROWID 最佳的优化方式是，如果业务允许，我们巧妙地消除这个动作的产生，但是如果存在有些非索引的字段必须展现，可是又不多的情况，我们该如何优化呢？”

我们前面在业务允许的情况下，将 `select * from t where object_id <= 5` 修正为 `select object_id from t where object_id <= 5` 从而消除回表，提升性能，假如有些字段必须展现，但又不多，该怎么办呢？

比如 `select object_id, object_name from t where object_id <= 5` 这个写法，非得展现 `object_name`，此时由于 `object_id` 索引不包含 `object_name` 的信息，回表获取 `object_name` 的动作势在必行，如下：

```
SQL> set autotrace traceonly
SQL> set linesize 1000
```

```
SQL> select object_id,object_name from t where object_id<=5;
执行计划
-----
Plan hash value: 2486998213

-----
| Id | Operation                                | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----
|  0 | SELECT STATEMENT                        |                |      4 |   316 |      3 (0)| 00:00:01 |
|  1 |  TABLE ACCESS BY INDEX ROWID          | T              |      4 |   316 |      3 (0)| 00:00:01 |
|*  2 |    INDEX RANGE SCAN                    | IDX1_OBJECT_ID |      4 |       |      2 (0)| 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
      2 - access("OBJECT_ID"<=5)
Note
-----
      - dynamic sampling used for this statement
统计信息
-----
          0  recursive calls
          0  db block gets
          5  consistent gets
          0  physical reads
          0  redo size
       553  bytes sent via SQL*Net to client
       400  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          4  rows processed
```

脚本 5-36 再观察一个 TABLE ACCESS BY INDEX ROWID 的例子

我们其实可以考虑在 object_id 和 object_name 列建组合索引，即可消除回表动作，如下：

```
SQL> create index idx_un_objid_objname on t(object_id,object_name);
索引已创建。
```

脚本 5-37 准备工作，对 t 表建联合索引

接下来我们发现，由于这个 idx_un_objid_objname 索引同时包含了 object_id 和 object_name 列的信息，所以返回 object_name 列时无须回表获取，性能总体比有回表的更高效，如下：

```
SQL> select object_id,object_name from t where object_id<=5
执行计划
```

```
-----
Plan hash value: 2827629532
-----
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		9	711	2 (0)	00:00:01
* 1	INDEX RANGE SCAN	IDX_UN_OBJID_OBJNAME	9	711	2 (0)	00:00:01

```
-----
Predicate Information (identified by operation id):
-----
```

```
1 - access("OBJECT_ID"<=5)
```

```
Note
```

```
-----
- dynamic sampling used for this statement
```

```
统计信息
```

```
-----
0 recursive calls
0 db block gets
3 consistent gets
0 physical reads
0 redo size
553 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
4 rows processed
```

脚本 5-38 联合索引消除了 TABLE ACCESS BY INDEX ROWID

大家注意到没，建联合索引后，回表的动作 TABLE ACCESS BY INDEX ROWID 在执行计划中压根儿找不到，3 个逻辑读及值为 2 的 COST 还是胜过有回表的 5 个逻辑读和值为 3 的 COST。

不过这里要注意平衡，如果联合索引的联合列太多，必然导致索引过大，虽然消减了回表动作，但是索引块变多，在索引中的查询可能就要遍历更多的 BLOCK 了，所以要全面考虑，联合索引不宜列过多，一般超过 3 个字段组成的联合索引都是不合适的。

同学们，我所说的回表的相关部分，大家听懂了吗？”

“听懂了！”从语气来看，同学们回答得很开心。

（3）聚合因子决定了回表查询的速度

“很好，大家要好好体会其中的奥秘，优化往往来自对细节的认知。这里我还要和大家说一个知识点，如果在 TABLE ACCESS BY INDEX ROWID 不可避免的情况下，必须执行这个回表

动作，是否回表查询方式也有效率高低之分呢？

实际上，回表查询的速度也是有差异的，这里引出一个重要概念叫**聚合因子**，下面我做系列试验，大家便会明白，这里我的试验方式有些特别，大家需要仔细听才能听明白。

首先我先构造两张简单的表：

```
SQL> drop table t_colocated purge;
表已删除。
SQL> create table t_colocated ( id number, col2 varchar2(100) );
表已创建。
SQL> begin
    for i in 1 .. 100000
    loop
        insert into t_colocated(id,col2)
        values (i, rpad(dbms_random.random,95,'*') );
    end loop;
end;
/
PL/SQL 过程已成功完成。
SQL> alter table t_colocated add constraint pk_t_colocated primary key(id);
表已更改。
SQL> drop table t_disorganized purge;
表已删除。
SQL> create table t_disorganized
    as
    select id,col2
    from t_colocated
    order by col2;
表已创建。
SQL> alter table t_disorganized add constraint pk_t_disorg primary key (id);
表已更改。
```

脚本 5-39 聚合因子试验准备，建有序和无序的表各一张

请大家先听我解释我构造的 t_colocated 和 t_disorganized 是何目的。在 t_colocated 表中，表的数据基本上是根据 id 从 1 到 100000 的顺序插入的，而我们都知，索引是有排列的，此时 id 列上的索引存放的数据也是按 1 到 100000 的顺序插入的。表和索引两者的排列顺序相似度很高，我们就称之为聚合因子比较低。

接下来看这个 t_disorganized 表，很显然由于表的插入顺序是依据 col2 这个插入记录为随机值的列来排序的，显然和有序的索引块 1 到 100000 的顺序有天壤之别。表和索引两者之间的排列顺序相似度差异明显，我们就称之为聚合因子比较高。

可以通过数据字典来判断索引的聚合因子情况，如下：

```
SQL> set linesize 1000
SQL> select index_name,
        blevel,
        leaf_blocks,
        num_rows,
        distinct_keys,
        clustering_factor
        from user_ind_statistics
        where table_name in( 'T_COLOCATED','T_DISORGANIZED');
```

INDEX_NAME	BLEVEL	LEAF_BLOCKS	NUM_ROWS	DISTINCT_KEYS	CLUSTERING_FACTOR
PK_T_COLOCATED	1	208	100000	100000	1469
PK_T_DISORG	1	208	100000	100000	99944

脚本 5-40 分别分析两张表的聚合因子情况

我们来说明一下 CLUSTERING_FACTOR 的官方解释：表明有多少临近的索引条目指到不同的数据块。

至此大家明白了 CLUSTERING_FACTOR 取值 99944 接近表记录的 100000，说明绝大部分的临近索引条目都指向了不同的数据块，这个聚合因子有多大啊。而 CLUSTERING_FACTOR 取值为 1469 说明总共只有 1469 个临近的索引条目指到了不同的数据块，总体还算不错。

听明白了吗？”梁老师没接着往下讲，想先了解一下大家的理解程度。

“梁老师，聚合因子我还是有些不太明白。”敬昱有些迷糊。

“没关系敬昱同学，其实我们也不要纠结太多，就记得表的插入顺序和索引列的顺序基本一致，从索引中回表查找数据块将会更容易查找就可以了，其实通俗地说就是索引块 A 里装 10 行列信息及 ROWID，这就可以理解为索引条目。

然后根据索引条目的 ROWID 找到表记录时，如果聚合因子很小，10 行索引条目可以全部在数据块 B 块中完整地找到。如果聚合因子很大，那郁闷了，或许这 10 行索引条目对应的数据块的 10 行记录，分布在 10 个不同的数据块里。那我们就要访问了 C 块，D 块，E 块等等，回表查询的性能当然就低了，听明白了吗敬昱？”

“明白了！”这下敬昱脑子清晰多了。

“好，我们试验继续了，大家都还清楚地记得我经常用如下方法来做性能跟踪，查看执行计划和逻辑读的情况，如下：

```
set linesize 1000
set autotrace trace
```

收获，不止 Oracle

```
--或者如下不显示语句返回结果，只跟踪
set autotrace traceonly
--执行你的 SQL
```

不过我现在准备教大家使用另外一个跟踪性能的工具，方法如下：

```
set linesize 1000
alter session set statistics_level=all;
---执行你的 SQL
SELECT * FROM table(dbms_xplan.display_cursor(NULL,NULL,'runstats_last'));
```

这种查看执行计划分析性能的方法将会在后续讲表连接时大量使用，这里先让大家熟悉一下，这方法和 set autotrace on 的差别大家日后自然会明白，不过这里有一个不方便的地方，那就是 alter session set statistics_level=all 的方法必须要让你的 SQL 执行完毕后才能 SELECT * FROM table(dbms_xplan.display_cursor(NULL,NULL,'runstats_last'))进行跟踪，如果你的 SQL 执行返回大量结果，屏幕将不停翻滚，这点要习惯。而 autotrace 的跟踪却有关闭显示的功能，比如 set autotrace off 即可关闭。

下面我们来分别统计 select /*+index(t)*/ * from t_colocated t where id>=20000 and id<=40000 和 select /*+index(t)*/ * from t_disorganized t where id>=20000 and id<=40000 的性能，这里的 HINT 是我要强迫 Oracle 使用索引，因为查询返回大部分记录，用索引反而低效，Oracle 会选择全表扫描的。我故意使用索引，就是要放大索引回表的次数，将性能的差距拉大，加深大家的印象。

首先观察 t_colocated 表的查询：

```
SQL> set linesize 1000
SQL> alter session set statistics_level=all;
会话已更改。
select /*+index(t)*/ * from t_colocated t where id>=20000 and id<=40000;
---此处略去 2 万行记录的输出刷屏过程
已选择 20001 行。
SQL> SELECT * FROM table(dbms_xplan.display_cursor(NULL,NULL,'runstats_last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID  fn73gtwh2dz98, child number 0
-----
```

```
select /*+index(t)*/ * from t_colocated t where id>=20000 and id<=40000
Plan hash value: 4204525375
```

```
-----
| Id | Operation                                | Name           | Starts | E-Rows | A-Rows |   A-Time | Buffers | Reads |
-----|-----|-----|-----|-----|-----|-----|-----|-----|
|  1 | TABLE ACCESS BY INDEX ROWID            | T_COLOCATED    |      1 |    21104 |    20001 | 00:00:00.22 |    2986 |    37 |
```

```
|* 2| INDEX RANGE SCAN |PK_T_COLOCATED| 1| 21104 | 20001 |00:00:00.16 | 1375 | 37|
PLAN_TABLE_OUTPUT
```

Predicate Information (identified by operation id):

```
2 - access("ID">=20000 AND "ID"<=40000)
```

Note

```
- dynamic sampling used for this statement
```

已选择 22 行。

脚本 5-41 首先观察有序表的查询性能

接下来观察 t_disorganized 表的查询：

```
SQL> SELECT * FROM table(dbms_xplan.display_cursor(NULL,NULL,'runstats_last'));
```

```
PLAN_TABLE_OUTPUT
```

```
SQL_ID 4fcff7124tj1j, child number 0
```

```
select /*+index(t)*/ * from t_disorganized t where id>=20000 and id<=40000
```

```
Plan hash value: 603192320
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	Reads
1	TABLE ACCESS BY INDEX ROWID	T_DISORGANIZED	1	26124	20001	00:00:02.59	21374	1115
* 2	INDEX RANGE SCAN	PK_T_DISORG	1	26124	20001	00:00:00.26	1375	37

PLAN_TABLE_OUTPUT

Predicate Information (identified by operation id):

```
2 - access("ID">=20000 AND "ID"<=40000)
```

Note

```
- dynamic sampling used for this statement
```

已选择 22 行。

脚本 5-42 再观察无序表的查询性能

比较发现，t_colocated 表的索引读产生的逻辑读为 2986，而 t_disorganized 表的索引读产生的逻辑读为 21374，差别居然达到近 10 倍。

同样大小的表和同样大小的索引，且记录数也相同，执行的是同样的语句，仅是聚合因子的

收获，不止 Oracle

差异，或者说是表的排列顺序的差异，居然导致性能差异达到 10 倍之多，如果不是亲眼所见，怕很多人都难以相信吧，怎么样，有意思吧。”梁老师笑了笑。

“太有趣了！”小莲激动地这么一喊，引来了一阵笑声。

“没完，老师还有问题要问大家。”梁老师又继续了，“请问，你们想不想让表里的所有索引列的聚合因子都比较低，从而提升回表的性能呢？”

“想啊！”同学们回答得还挺整齐的。

“可能吗？”梁老师问。

有陷阱！饱受梁老师忽悠的同学们忽然都警觉了，没人应话了。

10 秒过后，终于有人想明白了。“梁老师，不可能！列的索引的排序是按列的内容来排序的，各列里的内容各不相同，表只有一种插入顺序，如何去匹配全部这些呢？”最敏锐的总是小姑娘晶晶。

“很好，那我们怎么选择呢？我们当然是突出重点了，某列的读取频率远高于其他列，那就保证表的排列顺序和这列一致，按照这列的顺序，我们重组一下表记录来优化即可了。到此我把索引存储值的各个应用都说完了。”

小莲觉得这系列课程太实用了，而且完全可以有试验结果，可以再现，感觉受益颇多。

5.2.1.7 活用三特征之索引有序

1. ORDER BY 排序优化

“现在我们开始讲述索引的第三个特点，就是索引本身是有序的，这个特点有什么用呢？我们先来做一组试验看看。

```
SQL> set autotrace traceonly
```

```
SQL> set linesize 1000
```

```
SQL> drop table t purge;
```

表已删除。

```
SQL> create table t as select * from dba_objects;
```

表已创建。

```
SQL> set autotrace traceonly
```

```
SQL> select * from t where object_id>2;
```

已选择 55621 行。

执行计划

Plan hash value: 1601196873

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		53391	9228K	175 (1)	00:00:03

```
|* 1 | TABLE ACCESS FULL | T | 53391 | 9228K | 175 (1) | 00:00:03 |
```

Predicate Information (identified by operation id):

```
1 - filter("OBJECT_ID">2)
```

Note

```
- dynamic sampling used for this statement
```

统计信息

```

0 recursive calls
0 db block gets
4438 consistent gets
0 physical reads
0 redo size
2811132 bytes sent via SQL*Net to client
41188 bytes received via SQL*Net from client
3710 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
55621 rows processed
```

脚本 5-43 未排序前的性能开销（COST）较低

接下来我们执行一个类似的语句，差别就是增加了一个 order by 的关键字来排序。

```
SQL> select * from t where object_id>2 order by object_id;
```

已选择 55621 行。

执行计划

Plan hash value: 961378228

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		53391	9228K		2249 (1)	00:00:27
1	SORT ORDER BY		53391	9228K	21M	2249 (1)	00:00:27
* 2	TABLE ACCESS FULL	T	53391	9228K		175 (1)	00:00:03

Predicate Information (identified by operation id):

```
2 - filter("OBJECT_ID">2)
```

Note

```
- dynamic sampling used for this statement
```

统计信息

```
-----
0 recursive calls
0 db block gets
770 consistent gets
0 physical reads
0 redo size
2804640 bytes sent via SQL*Net to client
41188 bytes received via SQL*Net from client
3710 SQL*Net roundtrips to/from client
1 sorts (memory)
0 sorts (disk)
55621 rows processed
```

脚本 5-44 排序后的性能开销 (COST) 增大

大家仔细观察一下这两个语句产生的执行计划，说说有什么特点。”梁老师问大家。

“梁老师，没有 order by 语句的 SQL 没有产生排序，而有 order by 的 SQL 产生了排序，在 1 sorts (memory) 可以看到。”曾祥第一个起身回答。

“梁老师，没排序的语句的 COST 是 175，远低于排序语句的 2249，说明没排序的语句效率高，时间也是有差别的，没排序的显示完成时间是 3 秒，而有排序的显示完成时间是 27 秒。”敬昱也观察得很仔细。

“梁老师，很奇怪，有排序的逻辑读是 770，没排序的是 4438，居然有排序的 SQL 产生的逻辑读比没排序的要少得多，这是为什么啊？”晶晶发现新大陆了。

“大家说得非常好，这里说明了一个道理，大家要认真听，真正决定性能的是 COST 的高低和真实完成的时间，一般 COST 越小性能越高，Oracle 执行计划的选择就是由 COST 来决定的，这在体系物理结构学习中我们就描述过了。而时间也是非常简单的衡量的方式，完成时间越短性能越高。

而逻辑读方面，我们是作为参考，在绝大部分情况下（甚至可以说 90%以上的场合），逻辑读越少性能越快，但在这里却不适用了，排序算法有些特别，内部的机制导致性能和逻辑读关系不是太大，主要是消耗在 CPU 性能上，开销极大。此外如果 PGA 区无法容纳下排序的尺寸而进入磁盘排序，那将成为更大的性能杀手。

大家觉得有什么方法可以让 select * from t where object_id>2 order by object_id 语句的排序动作消除？”

“梁老师，刚才你的例子里 t 表没索引，是不是在 OBJECT_ID 列建索引就可以了，因为索引本身就排过序了，所以不需要排序，对吗？”晶晶问。

“很好，那我在刚才的基础上再建一个索引，看看排序是否能消除，先建索引如下：

```
SQL> create index idx_t_object_id on t(object_id);
```

索引已创建。

脚本 5-45 观察排序试验的建索引准备工作

大家都确认可以用上索引并消除排序吗？”

“梁老师，用不到索引，不能消除排序！”小莲好像忽然想到什么，起身打断了梁老师。

“为什么用不到？”梁老师笑着问。

“因为 OBJECT_ID 列并没有设置不允许空值！”

台下一片附和声。

“哦，看来想问题真全面啊，那让我们试验看看，如下：

```
set linesize 1000
set autotrace traceonly
SQL> select * from t where object_id>2 order by object_id;
```

已选择 55621 行。
执行计划

Plan hash value: 4285561625

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		55727	9632K	1035 (1)	00:00:13
1	TABLE ACCESS BY INDEX ROWID	T	55727	9632K	1035 (1)	00:00:13
* 2	INDEX RANGE SCAN	IDX_T_OBJECT_ID	55727		138 (1)	00:00:02

Predicate Information (identified by operation id):

2 - access("OBJECT_ID">2)

Note

- dynamic sampling used for this statement

统计信息

```

0 recursive calls
0 db block gets
8373 consistent gets
0 physical reads
0 redo size
2804640 bytes sent via SQL*Net to client
41188 bytes received via SQL*Net from client
3710 SQL*Net roundtrips to/from client
```

```
0 sorts (memory)
0 sorts (disk)
55621 rows processed
```

脚本 5-46 当排序列是索引列时，排序居然消除了

大家观察看看，有走索引吗？”梁老师问。

小莲有些傻了，自己想多了？

“哦，我，明白了！”小莲恍然大悟，“select * from t where object_id>2 order by object_id 当然不需要考虑空值问题，因为都已经说了条件是 object_id>2 的记录，空值已经排除了，这和写成 select * from t order by object_id 不一样啊。”

“说得非常好！”梁老师笑了，“自己弄明白了最好。”

“之前我让大家比较 select * from t where object_id>2 和 select * from t where object_id>2 order by object_id 两条语句在 t 表未建索引时的效率差别，让大家深刻感受到排序是影响性能的。

但是这两条语句毕竟是两条不等价语句，现在我们来观察 select * from t where object_id>2 order by object_id 这一条语句在建索引前后的执行效率，大家仔细观察，说说区别。”梁老师继续引导大家观察分析。

“梁老师，未建索引执行计划是全表扫描，如 1 sorts (memory)所示产生了一次排序，而建索引后执行计划是索引扫描，如 0 sorts (memory) 所示排序消失了。”晶晶飞快地回答了。

“梁老师，虽然建索引后语句逻辑读 8873 看上去比未建索引时的 770 大很多，但是建索引后的语句的实际执行时间是 13 秒，远快于未建索引时的 27 秒，且 COST 是 1035，也远低于未建索引时的 2249，所以建索引后的语句避免排序，效率是大大提升了。”经过梁老师的点拨，小莲对观察执行效率中哪几个地方也很敏感了。

“说得很好，现在大家知道了吧，原来索引真的可以有效避免排序，在 order by 列的语句中，我们就可以考虑在该列建一个索引来消除排序，尤其是当系统面临排序的严重瓶颈时。”梁老师总结到这里，忽然又让大家思考一个问题，“为什么消除排序后的逻辑读会有 8873 这么大呢？大家请注意观察并认真思考一下。”

“梁老师，我明白了，object_id>2 的条件导致返回了几乎所有的记录，在这种情况下用索引，效率往往是最低的，您说过了，因为全表扫描可以一次读取多个块，而索引的范围扫描一次只能读取一个块，且还要从索引中通过回表获取其他列的信息，需要读取的块就更多了！”看来小莲想明白了。

“说得非常好，还提及梁老师之前说的回表，很不错，其实梁老师在做这个试验的时候，并不能确认执行计划是否会走索引，我只能猜测说，有可能走索引，知道这是为什么吗？”

“因为这个语句用上索引必然导致增加了大量的逻辑读，唯一的优势就是消除了排序，需要优化器来综合判断计算 COST 的大小，所以梁老师心中也没有把握，是这样的吗梁老师？”晶晶

回答说。

“晶晶确实很机灵，说得非常对，我就是这么想的，不过如果语句是 `select object_id from t where object_id>2 order by object_id`，梁老师就增加更多的筹码了，对走索引更有自信了，这是为啥？”

“不用回表！”大家回答得异口同声。

“那我们看看试验的结果吧：

```
SQL> select object_id from t where object_id>2 order by object_id;
```

已选择 55621 行。

执行计划

Plan hash value: 2498590897

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		55727	707K	138 (1)	00:00:02
* 1	INDEX RANGE SCAN	IDX_T_OBJECT_ID	55727	707K	138 (1)	00:00:02

Predicate Information (identified by operation id):

1 - access("OBJECT_ID">2)

Note

- dynamic sampling used for this statement

统计信息

0	recursive calls
0	db block gets
3824	consistent gets
0	physical reads
0	redo size
806579	bytes sent via SQL*Net to client
41188	bytes received via SQL*Net from client
3710	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
55621	rows processed

脚本 5-47 消除 TABLE ACCESS BY INDEX ROWID 后 COST 更低了

减少了回表动作后，逻辑读从 8373 减少到 3824，COST 从 1035 减少到 138，执行时间从 13 秒减少到 2 秒，差别还真大啊。不过大家要切记一点，这个写法是不等价写法，`select` 某列和 `select*` 在需求上是有天壤之别的，写这个只是让大家再次牢记，避免回表是可以大大提升性能的，要善

于寻找可以避免回表的机会，千万别犯了业务允许只取一列而你却取了全部列这样的低级错误。”

2. DISTINCT 排重优化

“ORDER BY 语句的含义就是要排序，除此之外是否还有一些其他的语句需要排序呢？比如 DISTINCT 这个常见的排除重复记录的写法大家都见过吧，DISTINCT 会用到排序吗？能否考虑用索引来消除排序？”梁老师问。

“会，因为只有排过顺序后，才更容易去除重复记录，另外我觉得可以考虑用索引来消除排序，还是因为索引本身就排过序了。”小莲一下子就反应到了。

“说得很好，我们做试验观察一下 DISTINCT 是否能用到索引，首先构造记录大量重复的 t 表，让 t 表的 object_id 的取值只有 2 和 3 两种，如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t as select * from dba_objects;
表已创建。
SQL> alter table T modify OBJECT_ID not null;
表已更改。
SQL> update t set object_id=2;
已更新 55658 行。
SQL> update t set object_id=3 where rownum<=25000;
已更新 25000 行。
SQL> commit;
提交完成。
```

脚本 5-48 DISTINCT 测试前的准备

然后进行 autotrace 跟踪，查看 select distinct object_id from t 的执行计划如下：

```
SQL> set autotrace traceonly
SQL> select distinct object_id from t;
执行计划
-----
Plan hash value: 1793979440

-----
| Id | Operation                | Name | Rows  | Bytes | TempSpc | Cost (%CPU) | Time      |
-----
| 0 | SELECT STATEMENT          |      | 53822 | 683K  |          | 433  (1)| 00:00:06 |
| 1 |  HASH UNIQUE              |      | 53822 | 683K  | 2120K   | 433  (1)| 00:00:06 |
| 2 |    TABLE ACCESS FULL     | T    | 53822 | 683K  |          | 175  (1)| 00:00:03 |
-----

Note
-----
- dynamic sampling used for this statement
```

统计信息

```

-----
0 recursive calls
0 db block gets
771 consistent gets
0 physical reads
0 redo size
452 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
2 rows processed

```

脚本 5-49 发现 DISTINCT 会产生排序

大家说说，到底有没有产生排序？”梁老师问。

“梁老师，从 0 sorts (memory) 以及 0 sorts (disk)可以看出没有排序。”晶晶眼睛很尖，一下就看到了关键之处。

“梁老师，有排序！TempSpc 表示排序使用了 2MB 的空间啊。”小莲仔细观察了一下，有了新发现。

“大家都观察得很仔细，实际情况是，DISTINCT 是有排序的，只是在 AUTOTRACE 的 SORTS 关键字中不会显现而已，我们来看看，如果不加 DISTINCT 写法，性能上差别有多大：

```
SQL> set autotrace traceonly
```

```
SQL> select object_id from t ;
```

已选择 55658 行。

执行计划

```
-----
Plan hash value: 1601196873
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		53822	683K	175 (1)	00:00:03
1	TABLE ACCESS FULL	T	53822	683K	175 (1)	00:00:03

Note

```
-----
- dynamic sampling used for this statement
```

统计信息

```
-----
0 recursive calls
```


0	db block gets
4434	consistent gets
0	physical reads
0	redo size
749902	bytes sent via SQL*Net to client
41210	bytes received via SQL*Net from client
3712	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
55658	rows processed

脚本 5-50 SQL 去掉 DISTINCT 后，排序立即消失

很显然，时间从 6 秒缩减为 3 秒，COST 从 433 减少为 175，TempSpc 关键字也不显现了，说明 DISTINCT 会因为排序而影响性能，不过这里要注意一点，DISTINCT 采用的是 HASH UNIQUE 的算法，其实如果语句修改为 select object_id from t where object_id=2 这样的等值查询而非范围查询时，将产生 SORT UNIQUE NOSORT 的算法，会消除排序，如下：

SQL> select distinct object_id from t where object_id=2;

执行计划

Plan hash value: 588537493

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		30607	388K	323 (1)	00:00:04
1	SORT UNIQUE NOSORT		30607	388K	323 (1)	00:00:04
* 2	TABLE ACCESS FULL	T	30607	388K	175 (1)	00:00:03

Predicate Information (identified by operation id):

2 - filter("OBJECT_ID"=2)

Note

- dynamic sampling used for this statement

统计信息

0	recursive calls
0	db block gets
771	consistent gets
0	physical reads
0	redo size
413	bytes sent via SQL*Net to client

```

400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 5-51 DISTINCT 遇到等值查询时略显特殊

观察发现，执行计划从 SORT UNIQUE 转变为 SORT UNIQUE NOSOR，代价从 423 减少为 323，略有提升，但是无论怎么说，加上 DISTINCT 都是一种会影响性能的查询方式。

现在我要进入关键的议题了，索引能否消除 DISTINCT 带来的排序，试验如下：

```
SQL> create index idx_t_object_id on t(object_id);
索引已创建。
```

脚本 5-52 为 T 表的 object_id 列建索引

之前小莲认为 DISTINCT 会排序的想法是对的，小莲还认为建索引可以消除排序，我们试验看看她说的是否正确：

```
SQL> select /*+index(t)*/ distinct object_id from t ;
执行计划
```

Plan hash value: 503711260

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		53822	683K	380 (2)	00:00:05
1	SORT UNIQUE NOSORT		53822	683K	380 (2)	00:00:05
2	INDEX FULL SCAN	IDX_T_OBJECT_ID	53822	683K	122 (1)	00:00:02

Note

- dynamic sampling used for this statement

统计信息

```

-----
0 recursive calls
0 db block gets
111 consistent gets
0 physical reads
0 redo size
452 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client

```

```
0  sorts (memory)
0  sorts (disk)
2  rows processed
```

脚本 5-53 该列建索引，DISTINCT 引发的排序即可消除

这里显然从 SORT UNIQUE 转变为 SORT UNIQUE NOSORT，TempSpc 关键字不见了，说明排序消除了，COST 也从 433 减少为 380，我这里用 /*+index(t)*/ 是为了强制让 Oracle 走索引，来检验索引是否可以消除 DISTINCT 产生的排序。

看来小莲又猜对了。”梁老师看着小莲说。

“不过现实中，DISTINCT 语句靠索引来优化往往收效是不明显的，因为大多数情况用到 DISTINCT 都是因为表记录有重复，因此我们首要的是要考虑为什么重复，后面我会给大家上优化思想课程，里面就有类似的案例。”

3. 索引全扫与快速全扫

“请大家注意观察我在执行 `select /*+index(t)*/ distinct object_id from t` 时产生的执行计划中，索引的扫描方式是什么？”梁老师问。

“INDEX FULL SCAN。”小莲第一个回答。

“很好！现在请大家花点时间从头到尾翻阅梁老师的教材，把索引出现过的执行计划都找出来，说说都有哪些。”梁老师让大家细心地查找一遍。

“索引高度较低的章节中，用到的索引是 INDEX RANGE SCAN。”曾祥接着回答。

“COUNT(*) 优化中，用到索引的是 INDEX FAST FULL SCAN。”敬昱也发现了一个不同的。

“在 MAX/MIN 这部分教材中涉及 INDEX FULL SCAN (MIN/MAX)。”晶晶大声说道，“梁老师，这些执行计划的差异在您上 MAX/MIN 课程时我提问过了啊，只是当时您没回答我。”

“说得没错，确实是这些晶晶之前提出过的疑问，现在我来回答大家吧。

“MAX/MIN 就不用说了，这是针对最大最小取值的一种特别的索引扫描方式，而 INDEX RANGE SCAN 这个也简单，显然就是一种针对索引高度较低这个特性实现的一种范围扫描，在返回记录很少时相当高效。

真正需要多说的显然是 INDEX FULL SCAN 和 INDEX FAST FULL SCAN 的差别。先说两者的相同点，那就是，都是针对整个索引的全扫描，从头到尾遍历索引，而非局部的 INDEX FULL SCAN (MIN/MAX) 和 INDEX RANGE SCAN。

那差别在哪里呢，INDEX FAST FULL SCAN 比 INDEX FULL SCAN 多了一个 FAST，所以差别就是索引快速全扫描 INDEX FAST FULL SCAN 比索引全扫描 INDEX FULL SCAN 更快。”

台下听得都呆住了，这叫啥差别啊，不过大家很快意识到是开玩笑，大家都笑了。

“别笑，索引快速全扫描真的是比索引全扫描更快，而且它们确实又都是遍历所有的索引块，为什么觉得好笑呢？”梁老师还在装模作样。

“梁老师，如果是这样，INDEX FULL SCAN 岂不是要被淘汰了？虽然慢肯定有适用的场合。”晶晶问。

“说得好，什么时候使用什么样的技术是关键。我们先说说两者的差异吧，为什么 INDEX FAST FULL SCAN 会比 INDEX FULL SCAN 更快，那是因为索引快速全扫描一次读取多个索引块，而索引全扫一次只读取一个块，明白他们速度差异的原因了吗？”

“明白了，那为什么要分这两种读取方式呢？”小莲问。

“你再想想。”梁老师并不做回答。

看着前面产生 INDEX FULL SCAN 的例子和之前 COUNT(*) 产生 INDEX FAST FULL SCAN 的例子，小莲终于明白了，她迅速站起来大声说：“我知道了，一次读取多个块不容易保证有序，而一次读取一个块可以保证有序，因此在有排序的场合，INDEX FULL SCAN 的顺序读可以让排序消除，INDEX FAST FULL SCAN 虽然减少了逻辑读，但是排序这个动作却无法消除。”

听到这里，梁老师用鼓掌的方式来回复小莲的回答。

“所以说 COUNT(*) 和 SUM 之类的统计根本无须使用排序，一般都走 INDEX FAST FULL SCAN，而涉及排序语句时，就要开始权衡利弊，也许使用 INDEX FAST FULL SCAN 更快，也许使用 INDEX FULL SCAN 更快，梁老师也无法确定，由 Oracle 的优化器计算出成本来选择。

实际上，刚才 select /*+index(t)*/ distinct object_id from t 的例子我为什么要加 HINT，就是因为实际上 Oracle 此时可能会选择 INDEX FAST FULL SCAN，宁愿排序无法避免，但是减少逻辑读，因为这是综合考虑后优化器计算其代价的一个过程，大家看我继续试验如下，现在我把 HINT 去掉，如下：

```
SQL> set linesize 1000
```

```
SQL> set autotrace traceonly
```

```
SQL> select distinct object_id from t ;
```

执行计划

Plan hash value: 2729247865

Id	Operation	Name	Rows	Bytes	TempSpc	Cost(%CPU)	Time
0	SELECT STATEMENT		53822	683K		286 (2)	00:00:04
1	HASH UNIQUE		53822	683K	2120K	286 (2)	00:00:04
2	INDEX FAST FULL SCAN	IDX_T_OBJECT_ID	53822	683K		28 (0)	00:00:01

Note

- dynamic sampling used for this statement

统计信息

收获，不止 Oracle

```
0 recursive calls
0 db block gets
116 consistent gets
0 physical reads
0 redo size
452 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
2 rows processed
```

脚本 5-54 INDEX FAST FULL SCAN (可让逻辑读减少，但是无法消除排序)

大家发现了什么吗？TempSpc 为 2120K 表示用到了排序，但是代价却是 286，低于之前我强制用 HINT 的 380。

说明在此时 Oracle 判断用索引快速全扫描虽然不能避免排序，但是减少逻辑读的好处还是很大的，因此选择了 INDEX FAST FULL SCAN。而 INDEX FULL SCAN 虽然执行计划看不到 TempSpc 关键字，排序避免了，但是逻辑读增加得太多，还是不合算。

而这些，都不是老师能左右的，老师能左右的就是建一个索引，让 Oracle 来选择，否则 Oracle 就是巧妇难为无米之炊。

我们最后再看看下一个试验，这里再次证明了 Oracle 是需要权衡利弊而非我们能左右的。”

```
SQL> select object_id from t order by object_id;
```

已选择 55658 行。

执行计划

Plan hash value: 439494919

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	
0	SELECT STATEMENT		53822	683K	122 (1)	00:00:02	
1	INDEX FULL SCAN	IDX_T_OBJECT_ID	53822	683K	122 (1)	00:00:02	

Note

- dynamic sampling used for this statement

统计信息

```
0 recursive calls
0 db block gets
3821 consistent gets
```

```

0    physical reads
0    redo size
749902 bytes sent via SQL*Net to client
41210 bytes received via SQL*Net from client
3712  SQL*Net roundtrips to/from client
0    sorts (memory)
0    sorts (disk)
55658 rows processed

```

脚本 5-55 INDEX FULL SCAN (可消除排序，但逻辑读比索引快速全扫描多)

4. UNION 合并的优化

“前面老师说的都是针对最简单的单表查询，不过同学们可不要轻视简单的语句，理由有两点：第一，你连简单语句的调优都不会，那复杂的又如何调优呢？第二，复杂的语句就是由简单的语句组合而成的。

现在我说说两张单表合并的情况，大家都接触过 SQL，先说说 UNION 和 UNION ALL 的差别在哪里？”梁老师问。

“两者都是合并，但是 UNION 是合并后没有重复记录，而 UNION ALL 没有做这个筛选。”看来小莲对 SQL 还是挺熟悉的。

“非常正确！那这两种写法，需要排序吗？”梁老师问。

“UNION ALL 肯定不需要排序了，因为就是简单的合并啊，不过 UNION 会有去除重复记录的动作，和 DISTINCT 很相似，我觉得需要排序。”晶晶回答。

“回答得真好，完全正确！”梁老师笑着说，“我们先来观察一下 UNION 语句是否需要排序，如下：

```

SQL> drop table t1 purge;
表已删除。
SQL> create table t1 as select * from dba_objects;
表已创建。
SQL> alter table t1 modify OBJECT_ID not null;
表已更改。
SQL> drop table t2 purge;
表已删除。
SQL> create table t2 as select * from dba_objects;
表已创建。
SQL> alter table t2 modify OBJECT_ID not null;
表已更改。
SQL> set linesize 1000
SQL> set autotrace traceonly
SQL> select object_id from t1

```

```
2 union
3 select object_id from t2;
```

已选择 55661 行。
执行计划

Plan hash value: 3008085330

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		130K	1659K		978 (49)	00:00:12
1	SORT UNIQUE		130K	1659K	5153K	978 (49)	00:00:12
2	UNION-ALL						
3	TABLE ACCESS FULL	T1	69957	888K		175 (1)	00:00:03
4	TABLE ACCESS FULL	T2	60726	770K		175 (1)	00:00:03

Note

- dynamic sampling used for this statement

统计信息

0	recursive calls
0	db block gets
1542	consistent gets
0	physical reads
0	redo size
807111	bytes sent via SQL*Net to client
41210	bytes received via SQL*Net from client
3712	SQL*Net roundtrips to/from client
1	sorts (memory)
0	sorts (disk)
55661	rows processed

脚本 5-56 UNION ALL 是需要排序的

看来确实有产生排序，TempSpc 显示排序的尺寸为 5153K，晶晶同学猜得没错，那这个语句可以通过建索引优化吗？”

“可以！”大部分同学回答得不假思索。

“那我们试验看看，如下：

```
SQL> create index idx_t1_object_id on t1(object_id);
索引已创建。
SQL> create index idx_t2_object_id on t2(object_id);
索引已创建。
```

```
SQL> set autotrace traceonly
SQL> set linesize 1000
SQL> select  object_id from t1
      2  union
      3  select  object_id from t2;
已选择 55661 行。
执行计划
```

Plan hash value: 669167125

Id	Operation	Name	Rows	Bytes	TempSpc	Cost(%CPU)	Time
0	SELECT STATEMENT		130K	1659K		691 (48)	00:00:09
1	SORT UNIQUE		130K	1659K	5153K	691 (48)	00:00:09
2	UNION-ALL						
3	INDEX FAST FULL SCAN	IDX_T1_OBJECT_ID	69957	888K		31 (0)	00:00:01
4	INDEX FAST FULL SCAN	IDX_T2_OBJECT_ID	60726	770K		31 (0)	00:00:01

Note

- dynamic sampling used for this statement

统计信息

```

0 recursive calls
0 db block gets
260 consistent gets
0 physical reads
0 redo size
807111 bytes sent via SQL*Net to client
41210 bytes received via SQL*Net from client
3712 SQL*Net roundtrips to/from client
1 sorts (memory)
0 sorts (disk)
55661 rows processed
```

脚本 5-57 索引无法消除 UNION ALL 排序 (INDEX FAST FULL SCAN)

发现建了索引后 Oracle 会走快速全扫描，排序不可避免。”

“梁老师，这个 INDEX FAST FULL SCAN 是索引快速全扫描，本来就不能避免排序，如果是 INDEX FULL SCAN 索引全扫描就可以避免排序了！”小莲大声说。

“小莲真是火眼金睛啊！”梁老师笑了，“看来又是 Oracle 觉得排序开销的省去和逻辑读代价的减少不可兼顾而做出的选择，是吗？”

“是的！”小莲坚定地说。

收获，不止 Oracle

“好吧，我们看试验如下，我用 HINT 让 Oracle 走索引全扫描，如下：

```
SQL> select /*+index(t1)*/ object_id from t1
2 union
3 select /*+index(t2)*/ object_id from t2;
已选择 55661 行。
执行计划
```

Plan hash value: 243121257

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		130K	1659K		903 (48)	00:00:11
1	SORT UNIQUE		130K	1659K	5153K	903 (48)	00:00:11
2	UNION-ALL						
3	INDEX FULL SCAN	IDX_T1_OBJECT_ID	69957	888K		138 (1)	00:00:02
4	INDEX FULL SCAN	IDX_T2_OBJECT_ID	60726	770K		138 (1)	00:00:02

Note

- dynamic sampling used for this statement

统计信息

```
0 recursive calls
0 db block gets
248 consistent gets
0 physical reads
0 redo size
807111 bytes sent via SQL*Net to client
41210 bytes received via SQL*Net from client
3712 SQL*Net roundtrips to/from client
1 sorts (memory)
0 sorts (disk)
55661 rows processed
```

脚本 5-58 INDEX FULL SCAN 的索引依然无法消除 UNION ALL 排序

查看发现，这次虽然使用了 INDEX FULL SCAN，但是并没有消除排序，TempSpc 显示排序尺寸依然是 5153K，如此 INDEX FULL SCAN 不能消除排序，与索引快速全扫描相比，索引全扫描只显劣势而没有任何优势了，COST 代价是 903，比 691 高了不少，最终 Oracle 当然弃用索引全扫描方式，为什么排序不能消除呢？其实也很简单，这是两个不同的结果集的筛选，各自的索引当然无法奏效。

我为什么要说 UNION 这一小节，其实是要让大家知道例外的情况，比如这个场景用索引是无法消除排序的。我所经历的众多案例中，最常见的 UNION 的优化居然是把 UNION 改为 UNION ALL。在某些业务场景下，两个表根本就不可能有重复，却用 UNION 而不用 UNION ALL，这时我们要做的事情就是，将 UNION 修改为 UNION ALL。”

5.2.1.8 不可不说的主外键设计

“现在大家应该对索引的结构了解得比较深刻了吧，不过最重要的不是了解结构，而是我们一起从结构中推理出了经典实用的索引三大特点，并游刃有余地将三大特点巧妙地应用在各个常见 SQL 的优化中。

现在我想和大家探讨数据库设计中最常见的一种实用技术——主外键。主外键有三大特点：第一，主键本身是一种索引；第二，可以保证表中主键所在列的唯一性；第三，可以有效地限制外键依赖的表的记录的完整性。其中前两个特点和 CREATE UNIQUE INDEX 建立的唯一性索引基本相同。

如同前面我上课的风格一样，语法我是不会特别说明的，大家自己在试验的步骤中可以学习到。现在我们讨论几个主外键相关的实用经典例子，首先讨论外键上的索引，大家先看我做的一组试验，具体如下。

1. 外键上的索引与性能

以下脚本完成建有主键的 T_P 表和有外键的 T_C 表，并插入记录，分别是 3139 和 54957，具体如下：

```
SQL> drop table t_p cascade constraints purge;
表已删除。
SQL> drop table t_c cascade constraints purge;
表已删除。
SQL> CREATE TABLE T_P (ID NUMBER, NAME VARCHAR2(30));
表已创建。
SQL> ALTER TABLE T_P ADD CONSTRAINT T_P_ID_PK PRIMARY KEY (ID);
表已更改。
SQL> CREATE TABLE T_C (ID NUMBER, FID NUMBER, NAME VARCHAR2(30));
表已创建。
SQL> ALTER TABLE T_C ADD CONSTRAINT FK_T_C FOREIGN KEY (FID) REFERENCES T_P (ID);
表已更改。
SQL> INSERT INTO T_P SELECT ROWNUM, TABLE_NAME FROM ALL_TABLES;
已创建 3139 行。
SQL> INSERT INTO T_C SELECT ROWNUM, MOD(ROWNUM, 1000) + 1, OBJECT_NAME FROM ALL_OBJECTS;
已创建 54957 行。
```

```
SQL> COMMIT;
提交完成。
```

脚本 5-59 外键索引性能研究之准备

我们来分析探讨一下如下语句的执行效率：

```
SELECT A.ID, A.NAME, B.NAME FROM T_P A, T_C B WHERE A.ID = B.FID AND A.ID = 880;
```

本章节一直在探讨索引与单表查询的优化，忽然来个两表关联，是否有些不习惯？”老师笑着问。

“不会！”台下同学回答得很响亮。

“这语句也太简单了吧！”小莲心中暗想。

“现在我们跟踪一下相关执行计划和代价，具体如下：

```
SQL> set autotrace traceonly
SQL> set linesize 1000
SQL> SELECT A.ID, A.NAME, B.NAME FROM T_P A, T_C B WHERE A.ID = B.FID AND A.ID = 880;
已选择 55 行。
执行计划
-----
Plan hash value: 924601924

-----
| Id | Operation                                | Name          | Rows  | Bytes | Cost (%CPU)| Time      |
-----
| 0 | SELECT STATEMENT                        |               | 59    | 3540 | 86 (2)| 00:00:02 |
| 1 |   NESTED LOOPS                          |               | 59    | 3540 | 86 (2)| 00:00:02 |
| 2 |     TABLE ACCESS BY INDEX ROWID        | T_P           | 1     | 30    | 2 (0)| 00:00:01 |
|* 3 |       INDEX UNIQUE SCAN                 | SYS_C0066212 | 1     |       | 1 (0)| 00:00:01 |
|* 4 |     TABLE ACCESS FULL                  | T_C           | 59    | 1770 | 84 (2)| 00:00:02 |
-----

Predicate Information (identified by operation id):
-----
   3 - access("A"."ID"=880)
   4 - filter("B"."FID"=880)
Note
-----
   - dynamic sampling used for this statement
统计信息
-----
          0  recursive calls
          0  db block gets
       336  consistent gets
```

```

0   physical reads
0   redo size
2760 bytes sent via SQL*Net to client
433 bytes received via SQL*Net from client
5   SQL*Net roundtrips to/from client
0   sorts (memory)
0   sorts (disk)
55  rows processed

```

脚本 5-60 外键未建索引前的表连接性能分析

接下来观察一下建索引的情况，具体如下：

```
SQL> CREATE INDEX IND_T_C_FID ON T_C (FID);
```

索引已创建。

```
SQL> SELECT A.ID, A.NAME, B.NAME FROM T_P A, T_C B WHERE A.ID = B.FID AND A.ID = 880;
```

已选择 55 行。

执行计划

Plan hash value: 175510515

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		55	3300	58 (0)	00:00:01
1	NESTED LOOPS		55	3300	58 (0)	00:00:01
2	TABLE ACCESS BY INDEX ROWID	T_P	1	30	2 (0)	00:00:01
* 3	INDEX UNIQUE SCAN	SYS_C0066212	1		1 (0)	00:00:01
4	TABLE ACCESS BY INDEX ROWID	T_C	55	1650	56 (0)	00:00:01
* 5	INDEX RANGE SCAN	IND_T_C_FID	55		1 (0)	00:00:01

Predicate Information (identified by operation id):

3 - access("A"."ID"=880)

5 - access("B"."FID"=880)

Note

- dynamic sampling used for this statement

统计信息

```

-----
0   recursive calls
0   db block gets
64  consistent gets
0   physical reads
0   redo size
2760 bytes sent via SQL*Net to client

```

```
433 bytes received via SQL*Net from client
5 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
55 rows processed
```

脚本 5-61 外键建索引后的表连接性能分析

大家说说看，哪种效率更高？”老师问。

“老师，外键建索引后，逻辑读从 336 降低到 64，效率明显更高，不过这是为什么呢，对于索引来说，两表关联和单表有什么区别？”晶晶问。

“回答得很好，提问得也很好！”老师说，“晶晶的问题和表连接的 NESTED LOOPS 连接方式有关，老师先卖一个关子，先不说，在后续上表连接时同学们自然就明白了，同学们先把这个结论记下来。”

2. 外键索引与锁的避免

“看来外键建索引还真是很有用，虽然因为后续会在下一章节《表的连接》中描述而没有细说，但是性能有大幅度提升却是大家显而易见的。

实际上，并不只是性能的提升，在外键上建索引还能有效避免锁的竞争，但这点很多人却都不太了解，下面我在上节试验的基础上继续做试验，如下：

--首先开启会话 1

```
SQL> select sid from v$mystat where rownum=1;
SID
```

976

```
SQL> DELETE T_C WHERE ID = 2;
```

已删除 1 行。

--接下来开启会话 2，也就是开启一个新的连接

```
SQL> select sid from v$mystat where rownum=1;
SID
```

980

```
SQL> DELETE T_P WHERE ID = 2000;
```

已删除 1 行。

脚本 5-62 外键有索引，没有死锁情况产生

以上操作一切正常，没有发现被锁住的现象。

其实本来是很可能被锁住的，只是因为外键所在的列建了索引，所以才有幸避免了。现在我们分别进 SID=976 和 SID=986 的两个 SESSION，分别执行 **rollback** 做回退操作，然后将外键上

的索引删除，如下：

```
SQL> drop index IND_T_C_FID;
索引已删除。
```

脚本 5-63 外键索引先删除

现在我们重复之前的两个 SESSION 的删除操作，如下：

```
--首先开启会话 1
SQL> select sid from v$mystat where rownum=1;
SID
-----
976
SQL> DELETE T_C WHERE ID = 2;
已删除 1 行。
--接下来开启会话 2，也就是开启一个新的连接
SQL> select sid from v$mystat where rownum=1;
SID
-----
980
--然后执行如下进行观察
SQL> DELETE T_P WHERE ID = 2000;
--居然发现卡住半天不动了！
```

脚本 5-64 外键索引删除后，立即有锁相关问题

同学们有兴趣可以自行根据老师的脚本试验一下，这里我们惊奇地发现，T_P 这个主键所在的表，居然因为外键所在的表随便删除一条记录而导致 T_P 所在的表完全锁住，无法做任何 DML 更新操作，大家有兴趣可以试验 2000 以外的其他的任意取值，都是 HANG 住不动！

这是为什么呢？这里还是把悬念留给同学们自己去学习研究，大家先记住这个试验和结论。”

小莲听到这里，觉得有些神奇，主外键在目前的项目中可谓屡见不鲜，但是自己倒是从未注意过外键的索引居然这么重要。

3. 主外键约束简单证明

“刚才老师说的两点，估计很多工作多年的技术人员也没有注意到，大家先有一个印象，老师在后面的章节中还会提及。现在老师简要说明一下主外键最基本的一个功能，一句话总结：外键所在表的外键列取值必须在主表中的主键列有记录。

这其实很好理解，好比员工表和部门表两张。一个公司出现有某部门，但该部门没有员工是正常的，说明部门正在筹备中。但是如果出现某员工没有部门归属就是很不正常的，这就是必须要主外键约束的原因了。

收获，不止 Oracle

比如我们随便执行如下命令，就发现出错了：

```
SQL> DELETE T_P WHERE ID = 2;
```

```
DELETE T_P WHERE ID = 2
```

```
*
```

第 1 行出现错误：

```
ORA-02292: 违反完整约束条件 (LJB.FK_T_C) - 已找到子记录
```

脚本 5-65 删除主键所在表的记录失败

根据前面试验脚本 ALTER TABLE T_C ADD CONSTRAINT FK_T_C FOREIGN KEY (FID) REFERENCES T_P (ID) 可以知道 FID 是依赖于 T_P 的主键的，于是我们查询如下：

```
SQL> select count(*) from T_C WHERE FID=2;
```

```
COUNT(*)
```

```
-----  
55
```

脚本 5-66 外键关联表的对应记录没删除

原来有 55 条记录依赖于主表 ID=2 的记录，怪不得删除不了，执行如下命令删除外键中的 FID=2 的记录后，主表 T_P 即可轻易删除了。

```
SQL> delete from T_C WHERE FID=2;
```

```
已删除 55 行。
```

```
SQL> COMMIT;
```

```
提交完成。
```

```
SQL> DELETE T_P WHERE ID = 2;
```

```
已删除 1 行。
```

```
SQL> COMMIT;
```

```
提交完成。
```

脚本 5-67 外键所在表的相关记录删除后，操作成功

Oracle 提供的这些功能保证了多表记录之间记录的制约性，非常方便。这个小节很简单，应该都会理解吧，大家还有什么想法和疑问吗？”

“老师，这样手工删除很麻烦啊，要是在删除主键记录时自动删除外键所在的表的对应记录，那不是很方便吗？”晶晶忽然想到这点。

“问得太好了，Oracle 其实提供了这个功能，称之为级联删除。”老师回答。

4. 级联删除操作很方便

“其实操作的方法很简单，就是在原先增加外键的基础上增加 ON DELETE CASCADE 的关键字即可。具体如下操作，首先删除外键，然后根据 ON DELETE CASCADE 关键字重建外键，具

体如下：

```
SQL> alter table T_C drop constraint FK_T_C;
表已更改。
SQL> ALTER TABLE T_C ADD CONSTRAINT FK_T_C FOREIGN KEY (FID) REFERENCES T_P (ID) ON DELETE CASCADE;
表已更改。
```

脚本 5-68 级联删除设置

我们试验发现 T_C 表的 FID 有取值为 3 的记录 55 条，不过由于级联删除设置后，居然自动删除了 T_C 表的 55 条记录，具体试验如下：

```
SQL> SELECT COUNT(*) FROM T_C WHERE FID=3;
COUNT(*)
-----
55
SQL> DELETE FROM T_P WHERE ID=3;
已删除 1 行。
SQL> COMMIT;
提交完成。
SQL> SELECT COUNT(*) FROM T_C WHERE FID=3;
COUNT(*)
-----
0
```

脚本 5-69 果然可以通过级联自动删除

同学们，这个方法虽然简便，不过却要慎用，因为很多情况下，有些人删除主键表的记录是误删，本来可以经过数据库的‘ORA-02292: 违反完整约束条件’的错误提示而及时被避免，但是如果使用了这个级联删除，数据库将在无须要我们确认的情况下，直接把外键所在的表的相关记录删除得一干二净，这是很危险的，所以这个技术需要谨慎使用！”

晶晶若有所思地点点头，确实，经老师一提醒，她也认识到，这个技术确实有一定的风险。

5. 改造为主键简便方法

“同学们，我想再问大家一个问题，建主键的语法类似 ALTER TABLE T_P ADD CONSTRAINT T_P_ID_PK PRIMARY KEY (ID)，如果今天生产系统有一张大表的某字段符合主键的条件，没有重复记录，但是却只是一个普通索引，要改造为主键，该如何操作呢？”老师问。

“删除该列索引，然后建上主键。”小莲回答得最快。

“这个方法是可以，不过如果是生产中某张大表做这样的删索引和增加主键的操作，必然对生产系统产生很大的影响，一般只能选择在深夜暂停相关应用，进行操作。实际上，这个改造很简单，根本不需要删除索引，重建主键。

收获，不止 Oracle

因为建主键的动作其实就是建了一个唯一性索引，再增加一个约束，仅此而已，刚才我举的例子，完全可以很轻松地完成，具体见如下试验：

```
SQL> drop table t cascade constraints purge;
表已删除。
SQL> CREATE TABLE T (ID NUMBER, NAME VARCHAR2(30));
表已创建。
SQL> INSERT INTO T SELECT ROWNUM, TABLE_NAME FROM ALL_TABLES;
已创建 3139 行。
SQL> COMMIT;
提交完成。
SQL> CREATE INDEX IDX_T_ID ON t(ID);
索引已创建。
```

脚本 5-70 在表 T 的 ID 列建普通索引

接下来我们完全可以很轻松地在 ID 列建上索引，直接如下操作即可：

```
SQL> alter table t add constraint t_id_pk primary key (ID);
表已更改。
```

脚本 5-71 为 ID 列增加主键约束，即成主键

小莲同学，听明白了吗？”

小莲点了点头，这下她彻底明白主键的建立步骤了，其实就是生成一个与外键相关的约束及默认产生一个能保证唯一的索引，所以刚才的步骤根本不需要像自己说的这么麻烦。

5.2.1.9 组合索引高效设计要领

“老师讨论了很长时间的单表与索引，忽然在主外键中说到了多表与索引，主要是为了避免大家听得太单一，睡着了。”老师笑着说。

同学们都乐了。

1. 适当的场合能避免回表

“不过我看大家好像还有点困的样子，那老师再来一点小改变。之前大多说的是单列上的索引，现在我们讨论一下多列上的索引，也就是联合索引，看看这其中有没有什么特别之处。”

“老师，关于联合索引，您前面不是已经说过了吗，在“巧用三特征之列值”中的“索引回表与优化”小节不是做过相关试验了吗？”记忆力很好的小莲忍不住打断了老师的话。

“是啊，具体章节是“访问字段不在索引列与组合索引”。”林君也补充了一句。

“说得真好，那你们觉得联合索引有什么用途呢？”

“避免回表！”同学们几乎是一起回答，看来大家都记起来了。

“那老师顺便多问一句，执行计划中，回表的动作叫什么？”

“TABLE ACCESS BY INDEX ROWID。”大部分同学都喊出来了。

2. 组合列返回越少越高效

“看来大家是真记住了，真掌握了，老师很欣慰。”梁老师开心地笑了，“不过联合索引并不只是避免回表的，很多时候，在 a 字段上查询返回的记录比较多，在 b 字段上查询返回的字段也比较多，如果 a 和 b 字段同时查询，返回的记录比较少，那就适合建联合索引了。

很显然类似 `select * from t where a=1 and b=2` 就是属于这种情况，如果在 a 和 b 字段建联合索引是不可能消除回表的，因为返回的是所有字段，但是只要 ‘a=某值’ 返回较多，‘b=某值’ 返回也较多，组合起来返回很少，就适合建联合索引。

不过老师始终强调一点，过多的字段建联合索引往往是不可取的，因为这样索引也必然过大，不仅影响了定位数据，更严重影响了更新性能，一般不宜超过 3 个字段组合。”

“老师，我怎么知道两个组合返回记录是否很少呢？”敬昱问。

“你可以用类似如下的方式测试一下啊：

```
Select count(*) from t where a=1;
Select count(*) from t where b=2;
Select count(*) from t where a=1 and b=2;
```

如果 a=1 and b=2 的返回和前面的单独 a=1 或者单独 b=2 的返回记录数差别不大，那组合索引的快速检索就失去意义了，单独建某列索引更好，因为单独建立的索引体积比组合索引要小，检索的索引块也更少。

除非这时是仅需要展现索引所在的列，如 `select a,b from t where a=1 and b=2`，那这联合索引就是回表的范畴，另当别论。”

“明白了，谢谢老师！”

3. 组合两列谁在前更合适

“现在老师要和大家讨论的是一个应用设计优化中非常重要的技术细节，组合索引两列，谁在前更合适？现在有一个普遍流行的观点就是，重复记录少的字段列放在前面，重复记录较多的字段列放在后。大家觉得对吗？”老师问。

同学们有点没整明白啥意思，都没有回答。

“看来老师说的不够清楚，那就做试验吧，在做试验的过程中大家就会明白老师的意思了，首先根据数据字典 `dba_objects` 构造 t 表，如下：

```
SQL> drop table t purge;
Table dropped
SQL> create table t as select * from dba_objects;
```

收获，不止 Oracle

其中 dba_objects 数据字典中包含两个字段，object_id 和 object_type，分别表示对象的编号和对象的类型，大家觉得哪个字段重复记录会更多？”老师问。

“肯定是 object_type 重复记录多了，对象类型一般就是表、索引、视图、同义词等等，没多少个吧，而对象编号应该是唯一的吧，几乎不重复吧！”晶晶回答。

“那大家重新体会一下老师刚才说的话，回答一下组合索引中，是否是重复记录少的字段列放在前面，比如针对 select * from t where object_id=20 and object_type='TABLE'这样的查询，如下两个索引，哪个更高效：

```
SQL> create index idx1_object_id on t(object_id,object_type);
Index created
SQL> create index idx2_object_id on t(object_type,object_id);
Index created
```

现在大家彻底明白老师的意思了吧？”

“老师，明显是第一种更高效！”小莲不假思索地回答，“因为如果是在 object_id 和 object_type 列分别建两个索引，肯定是 object_id 上的列有用，每次 where object_id=xxx 只会返回 1 条，属于相异基础很低的情景，索引读的性能很高。而 where object_type=xxx 将会返回非常多条，属于相异基础很高的情景，不适合用索引，所以我觉得联合索引应该是 object_id,object_type 的顺序最适合。”

(1) 等值查询情况下效率一样

“小莲同学说的到底对不对呢？老师不做回答，先准备系列试验来证明一下，我们用 HINT 分别固定让 select * from t where object_id=20 and object_type='TABLE' 这个语句走不同的两个索引，然后用 set autotrace 的手法来跟踪它们之间的差异，具体如下：

```
set autotrace traceonly
set linesize 1000
----测试 object_id,object_type 顺序的 idx1_object_id 索引是否高效
select /*+index(t,idx1_object_id)*/ * from t where object_id=20 and object_type='TABLE';

----测试 object_type,object_id 顺序的 idx2_object_id 索引是否高效
select /*+index(t,idx2_object_id)*/ * from t where object_id=20 and object_type='TABLE';
```

观察使用 **idx1_object_id** 索引情况下的执行计划，如下：

```
set autotrace traceonly
set linesize 1000

SQL> select /*+index(t,idx1_object_id)*/ * from t where object_id=20 and object_type='TABLE';
执行计划
-----
```

Plan hash value: 2486998213

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	93	2 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	T	1	93	2 (0)	00:00:01
* 2	INDEX RANGE SCAN	IDX1_OBJECT_ID	1		1 (0)	00:00:01

Predicate Information (identified by operation id):

2 - access("OBJECT_ID"=20 AND "OBJECT_TYPE"='TABLE')

统计信息

```

0 recursive calls
0 db block gets
4 consistent gets
0 physical reads
0 redo size
1198 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 5-72 观察 object_id,object_type 顺序的组合索引

观察使用 **idx2_object_id** 索引情况下的执行计划，如下：

set autotrace traceonly

set linesize 1000

SQL> select /*+index(t,idx2_object_id)*/ * from t where object_id=20 and object_type='TABLE';

执行计划

Plan hash value: 1913591113

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	93	2 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	T	1	93	2 (0)	00:00:01
* 2	INDEX RANGE SCAN	IDX2_OBJECT_ID	1		1 (0)	00:00:01

Predicate Information (identified by operation id):

```
2 - access("OBJECT_TYPE"='TABLE' AND "OBJECT_ID"=20)
统计信息
```

```
-----
0   recursive calls
0   db block gets
4   consistent gets
0   physical reads
0   redo size
1198 bytes sent via SQL*Net to client
400  bytes received via SQL*Net from client
2   SQL*Net roundtrips to/from client
0   sorts (memory)
0   sorts (disk)
1   rows processed
```

脚本 5-73 观察 object_type, object_id 顺序的组合索引

好了，大家比较一下，有啥发现？”老师问。

“老师，CONSISTENT GETS 和 COST 代价都是相同的，说明重复度低的列无论在前在后都一样！”小莲因为关注自己的回答是否正确，所以看得特别仔细。

“那就是说，小莲回答错误了啊。”老师笑了。

小莲也不好意思地笑了，看来是自己想当然了。

“其实刚才 object_id 和 object_type 列的查询都是等值查询，这里老师说一个重要结论：**在等值查询情况下，组合索引的列无论哪一列在前，性能都一样。**

不过如果是范围查询，那就有差别了，大家可以继续观察老师的一组试验。”

“这会有差别？”小莲觉得有些奇怪，一时还没想明白。

(2) 范围查询情况下差异明显

“大家好，现在我给大家测试的脚本为：select * from t t where object_id>=20 and object_id<2000 and object_type='TABLE'，这个和之前的 object_id=20 不同了，我们仍然用 HINT 来固定这个语句分别走两个不同的索引，比较性能的差异，具体准备脚本如下：

```
set autotrace traceonly
set linesize 1000
----测试 object_id,object_type 顺序的 idx1_object_id 索引是否高效
select /*+index(t,idx1_object_id)*/ * from t where object_id>=20 and object_id<2000 and object_type
='TABLE';

----测试 object_type,object_id 顺序的 idx2_object_id 索引是否高效
select /*+index(t,idx2_object_id)*/ * from t where object_id>=20 and object_id<2000 and object_type
='TABLE';
```

观察使用 **idx1_object_id** 索引情况下的执行计划，如下：

```

set autotrace traceonly
set linesize 1000
SQL> select /*+index(t,idx1_object_id)*/ * from t where object_id>=20 and object_id<2000 and
object_type='TABLE';
已选择 309 行。
执行计划
-----
Plan hash value: 2486998213

-----
| Id | Operation                      | Name           | Rows  | Bytes | Cost (%CPU) | Time      |
-----
| 0 | SELECT STATEMENT                |                | 110   | 10230 | 10 (0)| 00:00:01 |
| 1 | TABLE ACCESS BY INDEX ROWID    | T              | 110   | 10230 | 10 (0)| 00:00:01 |
|* 2 | INDEX RANGE SCAN                | IDX1_OBJECT_ID | 110   |       | 8 (0)| 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
  2 - access("OBJECT_ID">=20 AND "OBJECT_TYPE"='TABLE' AND "OBJECT_ID"<2000)
      filter("OBJECT_TYPE"='TABLE')
统计信息
-----
          0 recursive calls
          0 db block gets
        60 consistent gets
          0 physical reads
          0 redo size
       12620 bytes sent via SQL*Net to client
         620 bytes received via SQL*Net from client
          22 SQL*Net roundtrips to/from client
          0 sorts (memory)
          0 sorts (disk)
         309 rows processed

```

脚本 5-74 此例中，用到 **idx1_object_id** 性能更低

观察使用 **idx2_object_id** 索引情况下的执行计划，如下：

```

set autotrace traceonly
set linesize 1000
SQL> select /*+index(t,idx2_object_id)*/ * from t where object_id>=20 and object_id<2000 and
2 object_type='TABLE';
已选择 309 行。

```

```

执行计划
-----
Plan hash value: 1913591113

-----
| Id | Operation                                | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----+-----+-----+-----+-----+-----+-----+
|  0 | SELECT STATEMENT                        |                |    110 | 10230 |       7 (0)| 00:00:01 |
|  1 |   TABLE ACCESS BY INDEX ROWID         | T              |    110 | 10230 |       7 (0)| 00:00:01 |
|*  2 |    INDEX RANGE SCAN                    | IDX2_OBJECT_ID |    110 |       |       2 (0)| 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
   2 - access("OBJECT_TYPE"='TABLE' AND "OBJECT_ID">=20 AND "OBJECT_ID"<2000)

统计信息
-----
          0  recursive calls
          0  db block gets
        55  consistent gets
          0  physical reads
          0  redo size
       12620  bytes sent via SQL*Net to client
          620  bytes received via SQL*Net from client
          22  SQL*Net roundtrips to/from client
           0  sorts (memory)
           0  sorts (disk)
         309  rows processed

```

脚本 5-75 此例中，用到 idx2_object_id 性能更高

试验结束了，大家说说有什么发现？”老师问。

“老师，(object_id, object_type) 顺序组合的 idx1_object_id 索引的逻辑读是 60，而 (object_type,object_id) 顺序组合的 idx2_object_id 索引的逻辑读是 55，难道是重复度低的不应该放在前面，这是怎么回事啊？”小莲有些糊涂了。

“这里老师再总结一个重要结论：组合索引的两列，当一列是范围查询，一列是等值查询的情况下，等值查询列在前，范围查询列在后，这样的索引才最高效！”

4. 哪列在前更合适原理探讨

“现在老师要问大家一个问题，为什么在组合索引两列都是等值查询时，无论哪一列前置，性能都一样。而组合索引两列有一列是范围查询时，必须要等值查询列在前才更高效，和列的重复度没有关系，这是为什么？”老师问。


台下静悄悄地没人回答，根据试验结果，大家把结论都记住了，不过为什么会这样，还真没人想明白。

“没人回答，看来大家没想清楚是什么原因，老师要仔细把这事给大家整明白了，请大家要非常仔细地听讲。

现在我们来分析一下某 t 表，其中 AREA_CODE 表示地区号，必然是大量重复的。而另一个字段 ID 为序列号，必然是重复度很少的，只是在不同的地区可能会有重复。

(1) 等值查询性能为啥一样

现在针对 `SELECT * FROM T WHERE AREA_CODE=591 AND ID=101` 这个查询我们来探讨一下，索引应该是 AREA_CODE+ID 的组合索引更高效，还是 ID+AREA_CODE 的组合索引更高效，首先我们观察 AREA_CODE+ID 的组合情况，如图 5-13 所示：



AREA_CODE	ID	ROWID
591	99	1111111
591	100	1100112
591	101	1199113
591	102	1132188
591	103	1111121
591	104	1111143
591	105	1111117
591	106	1111118
592	98	1111119
592	99	1111122
592	100	1111188
592	101	1111177
592	102	1111166
592	103	1111189
592	104	1111152

图 5-13 等值查询中 AREA_CODE + ID 的组合索引

重要说明：AREA_CODE+ID 这个索引有一个很显然的特征，就是按 AREA_CODE 进行排序，在 AREA_CODE 相同的情况下，按照 ID 进行二次排序，大家认真观察上图即可明白。

步骤①中根据索引定位的原理，很快定位到 AREA_CODE=591 和 ID=101 的这一行，接下来在步骤②中，根据 ROWID=1199113 去表中把对应行记录展现出来。接下来虽然 AREA_CODE=591，但是 ID=102，说明 ID=101 不可能再出现了，因为 ID 列也是有序的。因此


收获，不止 Oracle

Oracle 的查询到步骤③之后停止了前进的步伐，查询完毕。

接下来我们探讨 ID+AREA_CODE 这个组合的情况，具体如图 5-14 所示。

重要说明：ID+AREA_CODE 这个索引同样具有明显的特征，就是按 ID 列排序，在 ID 列相同的情况下，再按照 AREA_CODE 列进行二次排序，大家认真观察图 5-14 即可明白。

步骤①中根据索引定位的原理，很快定位到 ID=101 和 AREA_CODE=591 的这一行。接下来在步骤②中，根据 ROWID=1199113 去表中把对应行记录展现出来。接下来虽然 ID=101，但是 AREA_CODE=592 说明 AREA_CODE=591 不可能再出现了，因为 AREA_CODE 列也是有序的。因此 Oracle 的查询到步骤③之后停止了前进的步伐，查询完毕。”



ID	AREA_CODE	ROWID
99	591	1111111
99	592	1111122
100	591	1100112
100	592	1111188
101	591	1199113
101	592	1111177
102	591	1132188
102	592	1111166
103	591	1111121
103	592	1111189
104	591	1111143
104	592	1111152
105	591	1111117
106	591	1111118

图 5-14 等值查询中 ID + AREA_CODE 的组合索引

(2) 范围查询性能缘何不同

“刚才我们研究过为什么在类似 SELECT * FROM T WHERE AREA_CODE=591 AND ID=101 这样的等值查询时，无论索引是 AREA_CODE+ID 的组合还是 ID+AREA_CODE 的组合都一样，大家应该都理解了。

现在我们把查询从等值查询转换为范围查询，比如针对 SELECT * FROM T WHERE AREA_CODE=591 AND ID>=98 AND COL2<=105 这个查询我们来探讨一下，索引应该是 AREA_CODE+ID 的组合更高效，还是 ID+AREA_CODE 更高效呢？

首先我们观察 AREA_CODE+ID 的组合情况，如图 5-15 所示。

重要说明：步骤①中根据索引定位的原理，很快定位到 AREA_CODE=591 和 ID=99 的这第一行记录，接下来在步骤②中，在满足 AREA_CODE=591 这个固定值的前提下，不断定位到满足条件的 AREA_CODE=591 和 ID=100，AREA_CODE=591 和 ID=101 等记录，这期间根据 ROWID=1111111、1100112、1199113、1132188、1111121、1111143、1111117 分别去表中把对应行记信息获取到，将所需要的列返回给用户。

	AREA_CODE	ID	ROWID
步骤①	591	99	1111111
	591	100	1100112
	591	101	1199113
	591	102	1132188
	591	103	1111121
	591	104	1111143
	591	105	1111117
步骤③	591	106	1111118
	592	99	1111122
	592	100	1111188
	592	101	1111177
	592	102	1111166
	592	103	1111189
	592	104	1111152

图 5-15 等值查询中 AREA_CODE + ID 的组合索引

当查询到 AREA_CODE=591 和 ID=106 后，Oracle 发现没有必要再走下去了，因为 ID=105 不可能再出现了，因为 ID 列也是有序的。因此 Oracle 的查询到步骤③之后停止了前进的步伐，查询完毕。

接下来我们探讨 ID+AREA_CODE 这个组合的情况，具体如图 5-16 所示。

重要说明，步骤①中根据索引定位的原理，很快定位到 ID=99 和 AREA_CODE=591 这第一行记录，接下来在步骤②中，大家发现有一个惊人的不同之处！

由于 ID 列不是等值而是范围，在 ID>=98 AND COL2<=105 的条件下，AREA_CODE 定位到 AREA_CODE=592 后，前进的步伐居然无法停止，因为在下一个区域内，比如 ID=101 后，完全可能再出现 591，大家一看图 5-16 便明白了。

直到跳出 ID 到 105 的范围，遇到 ID=106 记录后，Oracle 这才停止了前进的步伐，在老师举的这个极端例子里，Oracle 也就遍历完全部记录了，而图 5-15 显然早早就停止了前进的步伐。

这下大家明白了吗？”

“听明白了！”大家脱口而出。小莲心中又感慨了一番，动脑分析问题，真是有用啊！

	ID	AREA_CODE	ROWID
步骤①	99	591	1111111
	99	592	1111122
	100	591	1100112
	100	592	1111188
	101	591	1199113
	101	592	1111177
	102	591	1132188
	102	592	1111166
	103	591	1111121
	103	592	1111189
	104	591	1111143
	104	592	1111152
	105	591	1111117
步骤③	106	591	1111118

图 5-16 等值查询中 ID 和 AREA_CODE 的组合索引

5. 组合查询 in 的经典改写

“真听懂了吗，老师要通过一个经典案例来考考大家，还是用刚才的 t 表例子，只是增加了 update t set object_id=rownum 的操作，具体原因后续大家会明白的，脚本为：

```
drop table t purge;
create table t as select * from dba_objects;
update t set object_id=rownum ;
UPDATE t SET OBJECT_ID=20 WHERE ROWNUM<=26000;
UPDATE t SET OBJECT_ID=21 WHERE OBJECT_ID<>20;
commit;
create index idx1_object_id on t(object_id,object_type);
```

脚本 5-76 in 的优化之建表准备

运行结果如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t as select * from dba_objects;
表已创建。
SQL> update t set object_id=rownum ;
已更新 55665 行。
SQL> UPDATE t SET OBJECT_ID=20 WHERE ROWNUM<=26000;
已更新 26000 行。
SQL> UPDATE t SET OBJECT_ID=21 WHERE OBJECT_ID<>20;
已更新 29665 行。
SQL> commit;
提交完成。SQL> create index idx1_object_id on t(object_id,object_type);
索引已创建。
```

脚本 5-77 in 的优化的数据准备

接下来老师再问问如下两种写法，是否等价？”

```
select /*+index(t,idx1_object_id)*/ * from t where object_type='TABLE' and object_id >= 20 and object_id <= 21;
select /*+index(t,idx1_object_id)*/ * from t where object_type='TABLE' and object_id in (20,21);
```

“等价！”同学们大声回答。

“不是吧，你们这些大学生，连初中的数学都忘记了吗？”老师笑着说。

“不等价！”喊出声音的同学都意识到了，`object_id >= 20 AND object_id <= 21` 是一条线，而 `in (20,21)` 是两个点，完全不等价！

“大家醒悟过来了啊。”老师笑着说，“不过从实际情况来看，由于 `update t set object_id=rownum`，而且根据我们业务的需求分析，很少有 ID 编号是有小数点的，所以我们可以认为上述两种写法是等价的。

现在请大家动脑筋思考一个问题，这两种写法在我们现实工作中，大多是等价的，但是，这两种写法性能一样吗？”梁老师开始考大家了。

“老师，我觉得 `object_id >= 20 AND object_id <= 21` 的写法性能应该低于 `object_id in (20,21)` 的写法。”又是晶晶打破了课堂的沉默。

“为什么呢？”老师问。

“老师，您这次例子执行计划固定了 `idx1_object_id` 的索引，表示是 `object_id,object_type` 组合的索引，等同于必须根据 `object_id` 从 20 到 21 全部遍历，而 `object_type` 即便搜索到了拼音排序大于 'TABLE' 的取值，也无法停止前进的步伐。

但是 `IN (20,21)` 就完全不同了，可以看作是 2 个等值查询，一个是 `object_id=20`，另一个是 `object_id=21`，分别是 2 个等值查询，就可以定位到记录后快速停止继续搜索。”

“说得很好，下面我们做个试验证明一下晶晶的论证是否正确，首先看 `object_id >= 20 AND`

收获，不止 Oracle

object_id <= 21 的写法在只有 object_id,object_type 列索引下的情况，如下：

```
set autotrace traceonly
set linesize 1000
SQL> select /*+index(t,idx1_object_id)*/ * from t where object_TYPE='TABLE' AND OBJECT_ID >= 20 AND
OBJECT_ID<= 21;
已选择 3180 行。
执行计划
-----
Plan hash value: 2486998213
-----
| Id | Operation                      | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----|-----|-----|-----|-----|-----|-----|
| 0  | SELECT STATEMENT                |                     | 2789  | 482K  | 514  (1)| 00:00:07 |
| 1  | TABLE ACCESS BY INDEX ROWID    | T                   | 2789  | 482K  | 514  (1)| 00:00:07 |
|* 2  | INDEX RANGE SCAN                | IDX1_OBJECT_ID     | 204   |       | 510  (1)| 00:00:07 |
-----
Predicate Information (identified by operation id):
-----
   2 - access("OBJECT_ID">=20 AND "OBJECT_TYPE"='TABLE' AND "OBJECT_ID"<=21)
       filter("OBJECT_TYPE"='TABLE')
Note
-----
   - dynamic sampling used for this statement
统计信息
-----
          0  recursive calls
          0  db block gets
       746  consistent gets
          0  physical reads
          0  redo size
    162517  bytes sent via SQL*Net to client
     2721  bytes received via SQL*Net from client
       213  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
     3180  rows processed
```

脚本 5-78 范围查询性能较低

接下来看在只有 object_id,object_type 列索引时，针对 OBJECT_ID IN (20,21)改写的情况，具体如下：

```
SQL> select /*+index(t,idx1_object_id)*/ * from t t where object_TYPE='TABLE' AND OBJECT_ID IN (20,21);
已选择 3180 行。
```

执行计划

Plan hash value: 3012115211

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		2789	482K	4 (0)	00:00:01
1	INLIST ITERATOR					
2	TABLE ACCESS BY INDEX ROWID	T	2789	482K	4 (0)	00:00:01
* 3	INDEX RANGE SCAN	IDX1_OBJECT_ID	1		3 (0)	00:00:01

Predicate Information (identified by operation id):

3 - access(("OBJECT_ID"=20 OR "OBJECT_ID"=21) AND "OBJECT_TYPE"='TABLE')

Note

- dynamic sampling used for this statement

统计信息

```

0 recursive calls
0 db block gets
600 consistent gets
0 physical reads
0 redo size
162517 bytes sent via SQL*Net to client
2721 bytes received via SQL*Net from client
213 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
3180 rows processed

```

脚本 5-79 范围查询改造为 IN 写法后，性能提升

大家仔细观察看看，有什么发现和疑问呢？”老师问。

“老师，我的判断是正确的，OBJECT_ID IN (20,21)的逻辑读是 600，而 OBJECT_ID >= 20 and OBJECT_ID <= 21 的逻辑读是 746，差别还是很明显的。”晶晶有些得意地回答。

“老师，我觉得有点奇怪啊，这个索引明显不应该用 object_id,object_type 的组合啊，应是 object_type,object_id 的组合才对啊，不符合您之总结的等值查询列在前，范围查询列在后的特性啊？”小莲有些疑惑不解。

“两位同学都说得非常好，尤其是小莲同学，记得老师的结论，也记得老师的分析思路，所以看出老师的失误，这个查询因为有 object_TYPE='TABLE'这个等值查询条件，肯定应该建成

object_type,object_id 的组合索引才最高效！

不过老师今天是故意的，首先这个用错索引列并不影响我们关于 IN 改写的性能优化研究。其次是因为我马上要转入下一个话题了：**组合索引设计中需要考虑单列的查询情况**。这里凸显了了解系统、熟悉项目背景的重要性，敬请期待。”

“真没想到，看似简单的索引，在老师的层层挖掘下，居然有这么多实用的技术可以应用！”小莲心中又起感慨。

6. 设计需考虑单列的查询

“同学们，刚才老师在描述组合索引时强调过，当两个索引列都是等值查询时，无论哪一列在前都无所谓，性能都一样。如果涉及两索引列中的一列是等值查询而另一列是范围查询时，等值查询列在前的组合索引更高效，老师用试验和理论分析都进行了证明，大家应该印象很深刻了。

不过现实应用是非常复杂的，比如以 `select * from t where object_type='TABLE' and object_id>= 20 and object_id<=21` 这个语句为例。明明是一个范围和等值结合的查询，此时显然应该是(object_type, object_id)的组合更高效，可实际应用中，老师偏偏只建了(object_id, object_type)的组合索引，大家知道为什么吗？”老师问。

台下半年没人回答，小莲也有点犯晕，这是为什么呢？

“没人回答啊，这说明一个问题，大家还缺少经验。其实答案很简单，我提示一下，你的项目应用中，难道 t 表对应的操作，就只有这个 `select * from t where object_type='TABLE' and object_id>= 20 and object_id<=21` 的查询吗？”老师开始启发大家了。

“老师，我知道了，因为说不定有不少语句又是要这样查询的：`select * from t where object_type=>xxx and object_type<=xxx and object_id=20`，而且这样的查询比之前的查询量要大得多，此时就适合 object_id 在前 object_type 在后了。”晶晶反应很快。

“说得太好了，不过为什么不考虑建 2 个索引呢？这样两种语句就可以高效利用到各自不同的索引了。”老师继续问。

“老师，我猜测是因为前一种查询很少，而调换顺序的组合索引对提升性能也没有特别明显，所以你不建了，因为多建一个索引必然需要多维护一个索引，更新是有额外开销的。”晶晶继续回答。

“回答得完全正确，看来晶晶思考问题的方式方法是深得老师真传啊！”老师开玩笑地说。

大家也都笑了。

“晶晶同学回答得非常好，不过实际工作应用中还需要考虑另外一种情况，那就是组合索引需要考虑单列的查询情况。”老师说。

“什么叫考虑单列的查询情况？”林君有些不解地问。

“老师解释一下，比如 `select * from t where object_id=2 and object_type='TABLE'` 这个语句是两列的查询，而 `select * from t where object_id=2` 就是单列的查询。现在我们要讨论的重点是，项目

应用中既有单列查询又有多列查询的情况下，索引设计的技巧，准备的试验脚本如下：

```

---试验 1
drop table t purge;
create table t as select * from dba_objects;
create index idx_object_id on t(object_id,object_type);
set autotrace traceonly
set linesize 1000
select * from t where object_id=19;

--试验 2
drop index idx_object_id;
--注意，下面这个索引和上一次建的索引顺序不同
create index idx_object_id on t(object_type, object_id);
select * from t where object_id=19;

```

上述脚本中索引如果是建成 `create index idx_object_id on t(object_id)` 那就简单了，`select * from t where object_id=19` 必然能用到索引！可是现在索引是 `create index idx_object_id on t(object_id,object_type)`，还能用到吗？我们看执行的情况如下：

```

SQL> drop table t purge;
表已删除。
SQL> create table t as select * from dba_objects;
表已创建。
SQL> create index idx_object_id on t(object_id,object_type);
索引已创建。
SQL> set autotrace traceonly
SQL> set linesize 1000
SQL> select * from t where object_id=19;
执行计划

```

Plan hash value: 2486998213

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	177	3 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	T	1	177	3 (0)	00:00:01
* 2	INDEX RANGE SCAN	IDX_OBJECT_ID	1		2 (0)	00:00:01

脚本 5-80 组合索引的前缀与单列索引一致

现在我们把这个索引删除了，重新建一个索引，名字也叫 `IDX_OBJECT_ID`，只是顺序颠倒为 **(object_type, object_id)**，试验如下，看看 `select * from t where object_id=19` 的查询能否用到索引：

```
SQL> drop index idx_object_id;
```

索引已删除。

```
SQL> create index idx_object_id on t(object_type, object_id);
```

索引已创建。

```
SQL> select * from t where object_id=19;
```

执行计划

Plan hash value: 1601196873

	Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
	0	SELECT STATEMENT		9	1593	175 (1)	00:00:03
*	1	TABLE ACCESS FULL	T	9	1593	175 (1)	00:00:03

脚本 5-81 组合索引的前缀与单列索引不一致

由此我们得出重要结论：**如果单列的查询列和联合索引的前置列一样，那单列可以不建索引，直接利用联合索引来进行检索数据。**

实际上，如果单列 `OBJECT_TYPE` 的取值不多，这个查询还是有可能用到索引的，这种扫描方式有些特殊，称之为跳跃索引（index skip scan），不过应用的场景不广，这里就不细说了。大家记住我的重要结论即可。

现在谁还能再补充一下晶晶同学的回答。”老师问。

“老师，如果系统中有大量类似 `select * from t where object_id=19` 这样的利用索引可以提高效率的单列查询，同时又有大量类似 `select * from t where object_id=2 and object_type='TABLE'` 的组合查询，就可以尽量保证组合索引的前置列是单列查询的列。”还是晶晶抢答了。

“很好，这就是要补充的了。”老师说。

“老师，如果是 `object_id` 单列再建一个索引，这个索引比 `object_id,object_type` 的组合要小，占用的块少，不是更高效？”林君问。

“那你的意思是，再建一个 `object_id` 列的索引，然后针对组合查询范围写法，再建一个倒置方向的组合索引，弄三个索引，各取所需，语句都高效了，是这意思吗？”老师问。

“是的！”林君不断点头。

“如果你的系统是一个仅供查询的只读系统，林君说的是一个好主意，可是现实中我们的系统是一个查询更新兼而有之的系统，为了三个语句高效，立即草率地建三个索引是不可取，因为

索引多了，更新是会受到很大的影响的。

老师是怎么取舍的呢？一般来说我会考虑这些语句执行的频率和重要性，如果是偶尔执行又不是非常重要的查询，我是不会考虑为这些 SQL 语句的查询性能提升而建特定的索引的。老师之前有过一次经典案例，就是将某表的索引从 12 个减少为 2 个，最后大幅度提升了该表的更新速度。但是这期间的工作量是很大的，需要捞取和这些表有关的各类查询，分析重要性和频率以及各个查询的相关联关系，什么叫相关联关系，比如 where object_id=19 和 where object_id=2 and object_type='TABLE'，大家可以理解吗？”老师问。

“茅塞顿开！”曾祥大喊一声。

大家开始被曾祥的这一声大吼给吓了一跳，随后都乐了。

5.2.1.10 变换角度看索引的危害

“任何事物都有两面性，有好的一面，也有坏的一面，比如烹饪需要火，没有火也就没有美味的享受，但是火灾却是让人不寒而栗的事，这在生活中并不少见。索引也是如此，索引可以让我们查询变得更加快捷，但是却会影响更新的性能，这需要我们好好去把握。

以下做一个各表不同索引个数情况下，比较插入性能差异的相关试验，老师准备相关脚本如下：

```
SQL> drop table t_no_idx purge;
表已删除。
SQL> drop table t_1_idx purge;
表已删除。
SQL> drop table t_2_idx purge;
SQL> drop table t_3_idx purge;
表已删除。
SQL> drop table t_n_idx purge;
表已删除。
SQL> create table t_no_idx as select * from dba_objects;
表已创建。
SQL> insert into t_no_idx select * from t_no_idx;
已创建 55666 行。
SQL> insert into t_no_idx select * from t_no_idx;
已创建 111332 行。
SQL> insert into t_no_idx select * from t_no_idx;
已创建 222664 行。
SQL> insert into t_no_idx select * from t_no_idx;
已创建 445328 行。
SQL> insert into t_no_idx select * from t_no_idx;
已创建 890656 行。
SQL> commit;
提交完成。
```

收获，不止 Oracle

```
SQL> select count(*) from t_no_idx;
COUNT(*)
-----
1781312
SQL> create table t_1_idx as select * from t_no_idx;
表已创建。
SQL> create index idx_1_1 on t_1_idx(object_id);
索引已创建。
SQL> create table t_2_idx as select * from t_no_idx;
表已创建。
SQL> create index idx_2_1 on t_2_idx(object_id);
索引已创建。
SQL> create index idx_2_2 on t_2_idx(object_name);
索引已创建。
SQL> create table t_3_idx as select * from t_no_idx;
表已创建。
SQL> create index idx_3_1 on t_3_idx(object_id);
索引已创建。
SQL> create index idx_3_2 on t_3_idx(object_name);
索引已创建。
SQL> create index idx_3_3 on t_3_idx(object_type);
索引已创建。
```

脚本 5-82 做索引个数与插入速度试验前的准备

以上脚本分别建了没有索引的 `t_no_idx` 表，有 1 个索引的 `t_1_idx` 表，有 2 个索引的 `t_2_idx` 表以及有 3 个索引的 `t_3_idx` 表，4 张表的记录都是 1 781 312 条，这看得明白吧？”老师问。

“明白！”

1. 索引越多插入明显慢得多

“很好，那老师分别在这 4 张表里插入 10 万条记录，看看在相同记录不同索引个数的情况下，插入的速度有什么差别，具体如下：

```
SQL> set timing on
SQL> insert into t_no_idx select * from t_no_idx where rownum<=100000;
已创建 100000 行。
已用时间: 00: 00: 05.26
SQL> insert into t_1_idx select * from t_1_idx where rownum<=100000;
已创建 100000 行。
已用时间: 00: 00: 15.43
SQL> commit;
提交完成。
已用时间: 00: 00: 00.00
```

```
SQL> insert into t_2_idx select * from t_2_idx where rownum<=100000;
已创建 100000 行。
已用时间: 00: 03: 07.54
SQL> insert into t_3_idx select * from t_3_idx where rownum<=100000;
已创建 100000 行。
已用时间: 00: 04: 40.18
```

脚本 5-83 索引个数与插入速度关系紧密

“同学们，有什么发现？”老师问。

“没有索引，插入最快，索引越多，插入越慢。”敬昱抢先回答。

“回答得很好，没有索引时插入 10 万条记录只用 5 秒就完成了，有一个索引插入速度立即慢了 3 倍，花了 15 秒，接下来有 2 个索引的情况就发生了骤变，花费了 3 分钟多，在有 3 个索引时插入的时间是近 5 分钟。

大家知道为什么会这样吗？”老师问。

“因为有了索引，更新了记录就更新了索引，就要维护索引那种有序排列的结构，开销很大。”晶晶迅速作出回答。

“晶晶的回答让我很满意，因为她还提了有序这个关键词，说得非常到位。”老师表扬了晶晶。

“接下来，我将会给大家做另一组有趣的试验，不过由于执行时间可能会很长，现在时间也差不多了，所以你们可以下课回家。等你们下午回到课堂，老师再把结果展现给大家看。”

在同学们的掌声中，老师结束了上午的课程。

2. 无序插入索引影响更惊人

下午上课时间到了，大家一眼看到了老师展现在大屏幕上的试验结果，如下：

```
SQL> set timing on
SQL> insert into t_no_idx select * from t_no_idx where rownum<=100000 order by dbms_random.random
已创建 100000 行。
已用时间: 00: 00: 01.98
SQL> commit;
提交完成。
已用时间: 00: 00: 00.02
SQL> insert into t_1_idx select * from t_1_idx where rownum<=100000 order by dbms_random.random;
已创建 100000 行。
已用时间: 00: 00: 48.62
SQL> commit;
提交完成。
SQL> insert into t_2_idx select * from t_1_idx where rownum<=100000 order by dbms_random.random;
已用时间: 00: 14: 39.92
SQL> commit;
```

```
SQL> insert into t_3_idx select * from t_3_idx where rownum<=100000 order by dbms_random.random;  
已创建 100000 行。  
已用时间: 00: 20: 13.40
```

脚本 5-84 有序插入影响更大

“有什么发现，和上午的试验有什么区别？”看到同学们目不转睛地看着大屏幕，老师问大家。

“老师，上午的各表插入 10 万条并记录的试验是没有打乱顺序的插入，而这次是 order by dbms_random.random，表示打乱顺序插入。”林君发现新大陆了。

“老师，随机无序插入 10 万条和之前的正常插入 10 万条速度有天壤之别啊，之前 2 个索引情况下插入速度是 3 分钟，现在居然是近 15 分钟，之前 3 个索引情况下插入速度是 4 分钟，现在是 20 分钟！”看来小莲才是真正发现了新大陆。

“大家知道为什么吗？真正原因就在于晶晶同学说的‘有序’，Oracle 插入新数据，必然导致索引变大，由于索引是有序的，所以这些新增的索引的键值必须插入到特定的位置，而不是随机排放，比如某索引块 C 块存放取值为 100~200 的键值，这时插入 120~130 的记录，这时这 10 条记录一定要和之前存放键值为 100~200 的索引块放在一起，由于 C 块已经装满了，所以 C 块周围就要扩出 C1 块，放新增的数据。不过这个数据可不是简单地把 120~130 的记录插入，而是重组，保证 C 到 C1 块里的记录是有序的！”

这里在 C 块附近扩展 C1 块和重组数据的动作是一个艰难的动作，是需要很大开销的，所以上午的试验发现有索引插入很慢。

下午为什么无序插入更是惊人的慢，因为原先 120~130 的插入有一定的顺序，可以完成准备工作后快速扩展新块和批量重组。而无序操作，批量梦想成泡影！”

同学们恍然大悟。

“现在请同学们做一下优化大师，比如刚才有三个索引的表的无序插入用时 20 分钟，谁能优化到 1 分钟以内完成呢？”老师笑着问大家。

大家你看看我，我看看你，没想明白该怎么做。

“看来我说优化大师，就把大家给吓住了，其实我始终认为优化就是来源于生活的思想，大家看老师是如何把三个索引的无序插入控制在 1 分钟以内的，请看试验如下：

```
SQL> set timing on  
SQL> create index idx_no_1          on t_no_idx(object_id);  
索引已创建。  
已用时间: 00: 00: 09.68  
SQL> create index idx_no_2          on t_no_idx(object_name);  
索引已创建。  
已用时间: 00: 00: 17.76
```

```
SQL> create index idx_no_3      on t_no_idx(object_type);
索引已创建。
已用时间: 00: 00: 12.68
```

脚本 5-85 假如索引最后建，可巧妙提升性能

老师试验做好了，请问，老师在有 3 个索引的 200 万条记录的大表情况下的插入速度，从原先的 20 分钟提升到了多少呢？”

“老师，您只是建索引啊，我没看出您是什么意思啊？”敬昱一脸的困惑。

“老师，我明白了，真的是优化到 1 分钟以内了啊！”小莲和晶晶几乎是同时回答。

“那优化后到底需要多少时间呢，敬昱同学还不明白啊。”老师继续问。

“老师，需要的时间是 00: 00: 01.98 加 00: 00: 09.68 加 00: 00: 17.76 加 00: 00: 12.68 总计 40 秒左右。”晶晶快速做出回答。

“敬昱同学，你听明白了吗？”

“明白了，1.98 秒是没有索引的插入情况，剩下的 9.68 秒和 17.76 秒以及 12.68 秒是插入完毕后建索引的时间，所以假如我把 t_3_idx 表的三个索引给删除了，然后插入完毕后再建索引，就可以比之前快得多！”敬昱想着晶晶的数字，再看看老师的试验，终于想明白了。

“大家明白就好，现在老师告诉大家一个经典的案例。电信计费每月月初都会出账，销账，涉及大量的表插入操作。速度非常缓慢，往往需要十多个小时完成。这些表的记录和老师刚才在笔记本电脑的数据库试验环境可是有天壤之别的，不是 100 多万而是千万甚至上亿，有的表索引的个数也远不止三个，虽然机器性能非常强劲，但是也还是非常缓慢。

后来老师建议先将索引失效，等插入完毕后再生效索引，结果速度有了大幅度提升，出账时插表环节的时间从原先的十多个小时缩减到 2 小时以内，大大提升了总体的出账速度。”

“原来做优化也挺简单的！”敬昱心里想着，忍不住说出口。

敬昱的话引发台下一片大笑。

3. 修改删除与插入略有差别

“刚才描述的是 DML 语句中的插入语句与索引的关系，那 UPDATE 和 DELETE 语句与索引的关系呢，是否也是如此呢？”老师问。

“是！”

“不对，还是有差别的，比如 update t set object_id=888 where object_id=9 这条语句，如果 where object_id=9 只返回少量的几条记录，而 t 表记录很大，显然这条语句是 object_id 列有索引高效。而且其他列有索引也不至于影响这类语句的性能，比如 object_type 列有索引，这条更新语句会更新到这列索引吗？”

“不会！”

“那仅对这条语句而言，再多的其他列建了索引，性能又何干呢？”老师笑着说。

同学们也都笑了。

“不过话说回来了，如果表中经常要更新绝大部分记录，此时索引失去了快速检索的用途了，那这种索引还是赶紧删了好。

接下来说的是 DELETE 语句，这个和 UPDATE 有什么差别？”老师问。

“删除是某行的所有列都删除了，所以如果索引太多，那需要维护的索引列就更多。”小莲回答。

“刚才小莲说得很好，如果某表建了过多的索引，删除语句实际上是更新了所有的索引，而不同于 update 语句，更新哪一列影响哪一个索引，所以索引过多对 delete 的影响显然大于对 update 的影响（这里顺带提一下，delete 删除索引列后，索引块中的相关需删除记录只是被打上一个删除标志而已，并没有真正删除）。

不过 delete from t where object_id=9 此类的语句，如果只删除很少的记录，肯定也是 object_id 列有索引时高效。

现在我来总结一下索引过多对三种更新语句的影响：

- ① 对 INSERT 语句负面影响最大，有百害而无一利，只要有索引，插入就慢，越多越慢！
- ② 对 DELETE 语句来说，有好有坏，在海量数据库定位删除少数记录时，这个条件列是索引列显然是必要的，但是过多列有索引还是会影响明显，因为其他列的索引也要因此被更新。在经常要删除大量记录的时候，危害加剧！
- ③ 对 UPDATE 语句的负面影响最小，快速定位少量记录并更新的场景和 DELETE 类似，但是具体修改某列时却有差别，不会触及其他索引列的维护。

至此，老师终于说完了过多索引与 DML 更新语句的关系，同学们好好体会一下，学会多角度思考问题，不要只考虑单方面。”

台下大多同学都在不由自主地点头，看着同学们认真的样子，老师心里很欣慰。

4. 建索引动作引发排序及锁

“除了索引会影响更新语句外，建索引动作也需要谨慎，通过老师的课程，大家应该都知道建索引会排序了，排序是非常耗 CPU 的一个动作，如果在系统繁忙时再增加大量排序，对系统来说无疑是雪上添霜。

另外建索引的过程会产生锁，而且不是行级锁，是把整个表锁住，任何该表的 DML 操作都将被阻止。虽然建索引会产生锁这点老师之前没有细说过，不过聪明的人应该可以猜个八九不离十，现在我来考考大家，谁知道原因，为什么建索引会产生全表锁？”老师问。

“我知道，因为建索引是需要把当前索引列的列值都取出来，排序后依次插入块中形成索引块的，加上锁是为了避免此时列值被更新，导致顺序又变化了，影响了建索引的工作。”晶晶略微思考后做出了回答。

“说得很好，猜测得非常到位！不过这期间还有一些细节，关于建索引与锁的关系，将会在锁的这个章节中详细描述，为了加深印象，老师说说自己相关案例吧。

有一次电信某生产系统忽然遇到性能问题，前台登录缓慢，短信延迟，持续 15 分钟后系统恢复正常。事后介入调查分析才知道，原来在这个时间点，某 DBA 对某大表建了一系列索引，由于担心排序尺寸过大时间过长，该 DBA 居然用了 10 个并行度来建索引。接下来的重大后果就是 10 个 CPU 资源被耗尽，系统遇到严重的性能问题，直至 15 分钟后，索引建完，系统恢复正常。

由于 15 分钟的性能故障导致电信该系统遭遇客户投诉，该 DBA 也遭到了扣绩效的惩罚，这对我们来说，是一个经验教训，希望大家都记住。

接下来，老师还要说一个事，也是和建索引有关，这次结果更糟糕。同样是电信相关系统的一个案例，又是某 DBA 对大表建索引。为什么说此次更糟？因为上个案例的表恰好是极少更新的表，所以锁的问题没有影响到程序，而接下来的案例就和锁有关了，由于这个表记录极大，建索引整整建了 2 个小时，这期间的应用程序对该表的所有更新动作都由于锁等待而全部挂起无法执行，直至索引建完为止。

这下不只是慢的问题了，而是这两小时内与该表相关的更新操作全部无法成功，所有与该应用程序相关的应用模块都受到影响，严重影响了系统的使用，该故障影响相当大，被定位为一级故障。”

听了老师描述的真实案例，同学们都张大了嘴巴、瞪圆了眼睛，心中都暗自想：还好老师早说，以后打死我也不敢在大白天随便乱建索引。

5.2.1.11 如何合理控制索引数量

“听完了前面的内容，大家对索引的看法应该是经历了从盲目崇拜到冷静看待，越发全面了吧？”老师调侃着说。

还真是有这个感觉，小莲对老师的话有些共鸣。

“索引的危害大家都已经领教过了，那大家怎么知道哪些索引有用，哪些索引没用，不需要建呢？”老师问。

“老师，建了以后从来就没有被使用过的索引，肯定就是没用的，只会影响更新的速度！”小莲大声回答。

“说得太好了，不过小莲同学，你怎么知道哪些索引建了以后就从未被用过？”老师问。

大家思考了一会儿，没人能答上来。

“实际上 Oracle 提供了这种技术，方法如下：

----对需要跟踪的索引进行监控

```
alter index 索引名 monitoring usage;
```

----通过观察 v\$object_usage 进行跟踪

```
select * from v$object_usage;
```


收获，不止 Oracle

具体方法大家可以参考我的如下试验步骤。

首先建 t 表并建立对应索引，观察 v\$object_usage，发现并没有相关记录：

```
SQL> drop table t purge;
表已删除。
SQL> create table t as select * from dba_objects;
表已创建。
SQL> create index idx_t_id          on t (object_id);
索引已创建。
SQL> create index idx_t_name       on t (object_name);
索引已创建。
---未监控索引时，v$object_usage 查询不到任何记录
SQL> select * from v$object_usage;
未选定行
--接下来对 idx_t_id 和 idx_t_name 两列索引做监控
```

脚本 5-86 索引监控的查询脚本

接下来我们对两列索引进行监控，继续观察 v\$object_usage，发现有记录了，如下：

```
SQL> alter index idx_t_id nomonitoring usage;
索引已更改。
SQL> alter index idx_t_name nomonitoring usage;
索引已更改。
SQL> set linesize 166
SQL> col INDEX_NAME for a10
SQL> col TABLE_NAME for a10
SQL> col MONITORING for a10
SQL> col USED for a10
SQL> col START_MONITORING for a25
SQL> col END_MONITORING for a25
SQL> select * from v$object_usage;
INDEX_NAME TABLE_NAME MONITORING  USED          START_MONITORING  END_MONITORING
-----
IDX_T_ID    T          YES          NO            10/16/2012 16:58:12
IDX_T_NAME  T          YES          NO            10/16/2012 16:58:13
```

脚本 5-87 索引监控的实施

其中 USED=NO 表示从监控以来没有被使用过，接下来我们用一下索引，比如执行如下查询命令后继续观察 v\$object_usage：

```
--以下查询必然用到 object_id 列的索引
```

```
SQL> select object_id from t where object_id=19;
```

```
OBJECT_ID
```

```
-----
19
```

--观察分析，果然发现 IDX_T_ID 列的索引的 USED 果然更改为 YES

```
SQL> select * from v$object_usage;
```

INDEX_NAME	TABLE_NAME	MONITORING	USED	START_MONITORING	END_MONITORING
-----	-----	-----	-----	-----	-----
IDX_T_ID	T	YES	YES	10/16/2012 16:58:12	
IDX_T_NAME	T	YES	NO	10/16/2012 16:58:13	

脚本 5-88 索引监控的跟踪

现在大家观察一下，有什么变化啊？”老师问。

“IDX_T_ID 索引的 USED 变为 YES 了，说明被使用过了！”这下被曾祥抢先回答了。

“很好，如果你跟踪了某些重要表的部分索引，发现跟踪了一个月后执行 `select * from v$object_usage where USED='NO'` 有返回的，这些索引基本上是可以删除的了，因为表示一个月都从未被使用过！

不过这个技术也有不足之处，比如无法知道这些索引被执行过多少次，从而判断哪些索引是用得最少的，这些说不定也可作为删除的考虑对象。”老师总结说。

“老师，如果我想跟踪所有的索引，有方便的方法吗？”晶晶问。

“说得很好！你可以利用数据字典，比如 `user_indexes` 可以统计对当前用户下的所有索引，然后就可以批量写出来了。不过一般不需要对所有的索引进行监控，毕竟监控也是有代价的。”老师回答。

“老师，如果我不想监控呢，比如刚才那个我已经知道经常用了，那如何解除监控呢？”小莲忽然想到这个问题。

“问得好，这也是命令，很简单，只需把 `monitoring` 修改为 `nomonitoring`，即 `alter index 索引名 nomonitoring usage` 就实现了。

好了，关于监控索引就说到这里了，学习这小节完全就是因为索引有两面性，要全面考虑，老师经常用这一招在关键表中删除多余的设计不当的索引。”

小莲不住地点头，这下她觉得自己面对索引时很有一些得心应手的办法了。

5.2.2 位图索引的玫瑰花之刺

5.2.2.1 统计条数奋勇夺冠

“说到这里，老师基本上把索引最常见、最实用的技术说完了，有收获吗？”老师问。

“太有收获了！”曾祥大声地回应，显得有些捣蛋。

收获，不止 Oracle

“老师要考考你到底学了多少，COUNT(*)可以利用索引优化吗？”老师问。

“可以！只要表中有非空索引列，根据索引就可以获取到记录数，而索引一般情况下比表小得多，所以可以用索引来优化！”看来曾祥捣乱归捣乱，回答问题倒是不含糊。

“之前说的都是 BTREE 索引，现在我们要谈谈另外一种类型的索引：位图索引。这可是一个不一般的家伙，有着惊人的力量。关于它的原理我暂缓讲述，先做一系列试验来让大家见证一下它的威力，大家觉得意下如何？”老师问。

“好！”

“为了凸显威力，我构造的 t 表记录达到 300 多万条，在两个不同列先后建立普通索引和位图索引，然后比较性能差异。

首先构造 t 表，让 t 表记录有 300 多万条：

```
SQL> drop table t purge;
表已删除。
SQL> create table t as select * from dba_objects;
表已创建。
SQL> insert into t select * from t;
已创建 55654 行。
SQL> insert into t select * from t;
已创建 111308 行。
SQL> insert into t select * from t;
已创建 222616 行。
SQL> insert into t select * from t;
已创建 445232 行。
SQL> insert into t select * from t;
已创建 890464 行。
SQL> insert into t select * from t;
已创建 1780928 行。
SQL> update t set object_id=rownum;
已更新 3561856 行。
SQL> commit;
提交完成。
```

脚本 5-89 位图索引跟踪前准备

通过 set autotrace on 跟踪 COUNT(*)语句的执行计划及性能：

```
SQL> set autotrace on
SQL> set linesize 1000
SQL> select count(*) from t;
COUNT(*)
-----
```

```

3561856
执行计划
-----
Plan hash value: 2966233522
-----
| Id | Operation                | Name | Rows  | Cost (%CPU) | Time |
-----|-----|-----|-----|-----|-----|
|  0 | SELECT STATEMENT         |      |    1 |    10733   (1) | 00:02:09 |
|  1 |   SORT AGGREGATE         |      |    1 |              |       |
|  2 |    TABLE ACCESS FULL    | T     | 3063K |    10733   (1) | 00:02:09 |
-----

```

Note

```

-----
- dynamic sampling used for this statement
统计信息
-----

```

```

      0 recursive calls
      0 db block gets
48871 consistent gets
48543 physical reads
      0 redo size
    415 bytes sent via SQL*Net to client
    400 bytes received via SQL*Net from client
      2 SQL*Net roundtrips to/from client
      0 sorts (memory)
      0 sorts (disk)
      1 rows processed

```

脚本 5-90 观察 COUNT(*)全表扫描的代价

接下来在 object_id 列建索引，并设置该列属性为非空，测试 COUNT(*)的执行计划及性能：

```

SQL> create index idx_t_obj on t(object_id);
索引已创建。
SQL> alter table T modify object_id not null;
表已更改。
SQL> set autotrace on
SQL> select count(*) from t;
COUNT(*)
-----
3561856
执行计划
-----
Plan hash value: 278572740

```

Id	Operation	Name	Rows	Cost (%CPU)	Time
0	SELECT STATEMENT		1	1874 (2)	00:00:23
1	SORT AGGREGATE		1		
2	INDEX FAST FULL SCAN	IDX_T_OBJ	3063K	1874 (2)	00:00:23

Note

- dynamic sampling used for this statement
统计信息

0	recursive calls
0	db block gets
8330	consistent gets
8311	physical reads
0	redo size
415	bytes sent via SQL*Net to client
400	bytes received via SQL*Net from client
2	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
1	rows processed

脚本 5-91 观察 COUNT(*)用普通索引的代价

接下来我们在 t 表的 STATUS 列建一个位图索引，该列表示状态，一般只有有效和失效两种取值。建完索引后我们观察 COUNT(*)的执行计划和性能。

SQL> create bitmap index idx_bitm_t_status on t(status);

索引已创建。

SQL> select count(*) from t;

COUNT(*)

3561856

执行计划

Plan hash value: 4272013625

Id	Operation	Name	Rows	Cost (%CPU)	Time
0	SELECT STATEMENT		1	88 (0)	00:00:02
1	SORT AGGREGATE		1		

	2		BITMAP CONVERSION COUNT				3063K		88	(0)		00:00:02	
	3		BITMAP INDEX FAST FULL SCAN		IDX_BITMAP_T_STATUS								

Note

- dynamic sampling used for this statement

统计信息

```

0 recursive calls
0 db block gets
95 consistent gets
0 physical reads
0 redo size
415 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 5-92 观察 COUNT(*)用位图索引的代价

好了，老师在不同的前提条件下分别执行了三次 `select count(*) from t`，大家仔细观察看看有什么差异，能分析得出什么结论吗？”老师问。

“老师，第一次查询时由于 `t` 表没建任何索引，所以执行计划走的是全表扫描，代价是 **10733**，逻辑读是 **48871**；第二次查询时在 `t` 表的 `object_id` 列建了一个索引，执行计划是走 `object_id` 列的索引快速全扫描，代价是 **1874**，逻辑读是 **8330**；第三次在 `t` 表增加了一个索引，即 `status` 列的位图索引后，这个查询立即使用了这个位图索引，结果代价是 **88**，逻辑读是 **95**，这个位图索引居然比普通索引的逻辑读少了近百倍，比最早的全表扫描的逻辑读少了近千倍，分析 `COST` 代价的差异，也大致如此，真是难以想象，不可思议！”

因此我得出结论，**COUNT(*)**的性能，在非空列有 **BTREE** 索引的情况下，一般用到该索引性能远高于全表扫描。不过性能最高的却是列上有位图索引的情况，甚至比用到普通非空列的 **BTREE** 索引时的性能又高出一大截！”晶晶不仅是观察得仔细，回答得更详细。

“说得非常好，详细得老师都不知道该怎么补充了。”老师笑着说，“不过还有一个小细节，关于位图索引的，不知道谁还能发现。”

“我发现了，建位图索引时，并没有做类似 `alter table T modify status not null` 设置 `status` 列为非空属性的操作步骤，是否说明位图索引可以存储空值？”小莲回答得很迅速，原来她是早就注意到了。

“说得太好了，两位同学的回答让老师不知道该再补充什么了，只能说一句，两位同学说的，

都是对的。从这个语句的查询来看，位图索引查询不仅比 BTREE 索引快许多倍，连空值问题都无须考虑，大家是否觉得这个位图索引比 BTREE 索引更高级啊！”老师笑着问大家。

“是！”大家几乎是齐声回答，其中又是曾祥最为大声。

“那老师前面说了这么久的 BTREE 索引，都是低级技术了？”老师故意显示出很失望的表情。

大家都笑了，也明白是被老师忽悠了。

“请大家一定要牢记，只有最适合的技术而没有最高级的技术，不只是数据库领域，也不只是计算机领域，生活中也是如此，没有最好的老公，只有最适合的老公。”

最后一句话引发了台下一阵爆笑。

“大家别笑，后面的章节我会让大家更全面认识位图索引，除了让大家明白它在 COUNT(*) 大比拼中所向无敌、奋勇夺冠的秘密外，还要让大家知道它有哪些弱点，大家有兴趣了解吗？”

老师的这个开场试验一下子将同学们的积极性调到了顶点，老师辛苦讲述的 BTREE 索引在和位图索引的性能大比拼中显得如此不堪一击，真是神奇啊，其中到底有什么奥妙呢？

5.2.2.2 即席查询一骑绝尘

“刚才同学们已经察觉到位图索引勇猛过人，其实位图索引的抢眼之处并不局限于上述例子。现在我给大家举一个和人口普查应用相关的案例，在人口普查中，我们经常会需要做一些关于性别、年龄范围、出生地等多维度的分析统计，这类多维度的报表查询（如 `select * from t where col1=xxx and col2=xxx and col3=xxxx and col4=xxx...`）我们可以称之为即席查询。

首先我们构造 t 表如下，该表有性别、年龄范围及出生地等字段，具体如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t
(name_id,
 gender not null,
 location not null,
 age_group not null,
 data
)
as
select rownum,
       decode(ceil(dbms_random.value(0,2)),
              1,'M',
              2,'F')gender,
       ceil(dbms_random.value(1,50)) location,
       decode(ceil(dbms_random.value(0,3)),
              1,'child',
              2,'young',
```

```

        3,'middle_age',
        4,'old'),
    rpad('*',20,'*')
from dual
connect by rownum<=100000;
表已创建。

```

脚本 5-93 做位图索引与即席查询试验前的准备

在没有建任何索引时，我们发现全表扫描方式下 Oracle 的 COST 成本是 124，具体如下：

```

SQL> set linesize 1000
SQL> set autotrace traceonly
SQL> select *
  2  from t
  3  where gender='M'
  4  and location in (1,10,30)
  5  and age_group='child';
已选择 499 行。
执行计划
-----
Plan hash value: 1601196873
-----
| Id | Operation                | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----+-----+-----+-----+-----+-----+-----+
|  0 | SELECT STATEMENT         |      | 613   | 21455 |  124  (2)| 00:00:02 |
|* 1 |  TABLE ACCESS FULL      | T     | 613   | 21455 |    124  (2)| 00:00:02 |
-----+-----+-----+-----+-----+
Predicate Information (identified by operation id):
-----
   1 - filter(("GENDER"='M' AND ("LOCATION"=1 OR "LOCATION"=10 OR
        "LOCATION"=30) AND "AGE_GROUP"='child'))
Note
-----
   - dynamic sampling used for this statement
--略去统计信息部分

```

脚本 5-94 查询即席查询中应用全表扫描的代价

接下来我们建立三个列的联合索引，因为这三个列都是属于高度重复的列，而组合在一起返回是 499 条，因此可以考虑用组合索引试验看看，结果发现 Oracle 走的依然是全表扫描，具体如下：

```

SQL> create index idx_union on t(gender,location,age_group);

```


收获，不止 Oracle

索引已创建。

```
SQL> select *
```

```
2  from t
3  where gender='M'
4  and location in (1,10,30)
5  and age_group='child';
```

已选择 499 行。

执行计划

Plan hash value: 1601196873

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		613	21455	124 (2)	00:00:02
* 1	TABLE ACCESS FULL	T	613	21455	124 (2)	00:00:02

Predicate Information (identified by operation id):

```
1 - filter(("GENDER"='M' AND ("LOCATION"=1 OR "LOCATION"=10 OR
"LOCATION"=30) AND "AGE_GROUP"='child'))
```

Note

- dynamic sampling used for this statement

--略去统计信息部分

脚本 5-95 发现即席查询中，Oracle 不选择组合索引

这是为什么呢？我们强制用这个联合索引试验发现，Oracle 的代价居然是 39960，比用全表扫描的 124 高得多，实际上所有的代价都集中在 TABLE ACCESS BY INDEX ROWID 回表这个阶段，不过即便只是 INDEX RANGE SCAN 索引访问，代价也有 154，比全表扫描还高，因为该表总共也就 5 个字段，而索引就占了 3 个字段，索引的体积小的优势不明显，所以还不如全表扫描。

```
SQL> select /*+index(t,idx_union)*/ *
```

```
2  from t
3  where gender='M'
4  and location in (1,10,30)
5  and age_group='child';
```

已选择 499 行。

执行计划

Plan hash value: 306189815

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
----	-----------	------	------	-------	-------------	------

```

-----
| 0 | SELECT STATEMENT          |          | 613 | 21455 | 39960 (1) | 00:08:00 |
| 1 | INLIST ITERATOR            |          |     |      |           |          |
| 2 | TABLE ACCESS BY INDEX ROWID | T        | 613 | 21455 | 39960 (1) | 00:08:00 |
|* 3 | INDEX RANGE SCAN           | IDX_UNION | 60019 |      | 154 (0) | 00:00:02 |
-----

```

Predicate Information (identified by operation id):

```

3 - access("GENDER"='M' AND ("LOCATION"=1 OR "LOCATION"=10 OR "LOCATION"=30)
      AND "AGE_GROUP"='child')

```

Note

```

-----
- dynamic sampling used for this statement

```

--略去统计信息部分

脚本 5-96 强制即席查询使用组合索引性能更糟

接下来我们看看建立位图索引的情况，我们在 gender、location 和 age_group 三个字段分别建立了位图索引，然后观察查询的性能情况：

```

SQL> create bitmap index gender_idx on t(gender);
索引已创建。
SQL> create bitmap index location_idx on t(location);
索引已创建。
SQL> create bitmap index age_group_idx on t(age_group);
索引已创建。
SQL> select *

```

```

 2  from t
 3  where gender='M'
 4  and location in (1,10,30)
 5  and age_group='41 and over';

```

已选择 499 行。

执行计划

Plan hash value: 3513929889

```

-----
| Id | Operation                      | Name          | Rows  | Bytes | Cost (%CPU) | Time      |
-----
| 0 | SELECT STATEMENT                |               | 613   | 21455 | 11 (0)      | 00:00:01 |
| 1 | TABLE ACCESS BY INDEX ROWID    | T             | 613   | 21455 | 11 (0)      | 00:00:01 |
| 2 | BITMAP CONVERSION TO ROWIDS     |               |       |       |              |          |
| 3 | BITMAP AND                      |               |       |       |              |          |
| 4 | BITMAP OR                       |               |       |       |              |          |
-----

```

*	5	BITMAP INDEX SINGLE VALUE	LOCATION_IDX						
*	6	BITMAP INDEX SINGLE VALUE	LOCATION_IDX						
*	7	BITMAP INDEX SINGLE VALUE	LOCATION_IDX						
*	8	BITMAP INDEX SINGLE VALUE	AGE_GROUP_IDX						
*	9	BITMAP INDEX SINGLE VALUE	GENDER_IDX						

Predicate Information (identified by operation id):

5 - access("LOCATION"=1)
6 - access("LOCATION"=10)
7 - access("LOCATION"=30)
8 - access("AGE_GROUP"='child')
9 - access("GENDER"='M')

Note

- dynamic sampling used for this statement

--略去统计信息部分

脚本 5-97 即席查询应用到位图索引，性能有飞跃

我们惊奇地发现，COST 代价居然是 11，比起之前全表扫描的 124 有着天壤之别，比起联合索引的 39960 更是不可同日而语，这还只是针对 3 个字段的即席查询，如果是多个字段，差别更加明显了。

试验做完了，大家什么感觉啊，是不是除了惊奇，还是惊奇！”老师笑着问。

“位图索引真是个好东西啊！”曾祥忍不住大声喊了一句。

“是吗？”老师笑着说，“我看呆会儿曾祥同学看了老师接下来的系列试验后，看法就会改变了。”听到这里，同学们将身体坐得更直了，瞪大了眼睛期待着老师接下来的试验。

5.2.2.3 遭遇更新苦不堪言

“现在我就来看看，位图索引真的那么好吗？我们来看看下面一个有趣的试验，首先我们建立一个 SID=974 的连接，完成 t 表的简单插入，如下：

```
sqlplus ljb/ljb
SQL> select sid from v$mystat where rownum=1;
SID
-----
974
SQL> insert into t(name_id,gender,location ,age_group ,data) values (100001,'M',45,'child',rpad('*',20,'*'));
已创建 1 行。
```

脚本 5-98 位图索引遭遇锁困扰试验步骤 1

这里插入后暂且不提交，接下来新起一个 SID=975 的连接，继续另一条插入，结果插入被阻止，一动不动地卡住了，具体如下：

```
sqlplus ljb/ljb
SQL> select sid from v$mystat where rownum=1;
      SID
-----
      975
insert into t(name_id,gender,location ,age_group ,data) values (100002,'M',46, 'young', rpad('*',20,'*'));
--结果卡在这里一动不动了！
```

脚本 5-99 位图索引遭遇锁困扰试验步骤 2

那我们再换一个 SID=971 的连接，试验一下另一条插入，发现又可以正常插入，具体如下：

```
SQL> select sid from v$mystat where rownum=1;
      SID
-----
      971
SQL> insert into t(name_id,gender,location ,age_group ,data) values (100003,'F',47, 'middle_age', rpad('*',20,'*'));
已创建 1 行。
```

脚本 5-100 位图索引遭遇锁困扰试验步骤 3

这里插入后依然暂且不提交，接下来新起一个 SID=964 的连接，继续另一条插入，结果插入被阻止，一动不动地卡住了，具体如下：

```
SQL> select sid from v$mystat where rownum=1;
      SID
-----
      964
SQL> insert into t(name_id,gender,location ,age_group ,data) values (100003,'F',48, 'old', rpad('*',20,'*'));
```

脚本 5-101 位图索引遭遇锁困扰试验步骤 4

大家看看在建立了位图索引后，系统变成这样了，使用起来方便吗？”老师问。

“不方便，不过我还是没看明白刚才为什么 SID=971 的插入又可以成功啊？”小莲有些大惑不解。

“同学们可以注意观察一下，看看自己能否发现规律。”老师笑着回答。

“我发现规律了，只要第一次插入的是 gender='M'的记录，未提交时，新的 SESSION 再插入 gender='M'的记录，就会卡住。插入 gender='F'的记录是可以的。换句话说就是 gender 取某个值的插入，未提交，新 SESSION 的插入记录的 gender 取同样的值，必然卡住。”晶晶找规律总是特别快。

收获，不止 Oracle

“说得很好，现在对于这个表的插入，几乎只能是单进程单用户操作了，因为该 gender 列仅是 M 男和 F 女两种取值，某 SESSION 插入该表的记录是男，任何其他 SESSION 和男有关的记录都不可能插入成功，这样的系统还可以支持高并发吗？”老师问。

“不能！”还是曾祥的声音最大，这下他认识更全面了。

“老师，你的这个例子是有三个列建位图索引啊，如果 location 和 age_group 两列也和 gender 列类似，那不是更新起来更举步维艰了？”小莲忽然想到。

“说得非常好，老师刚才的试验例子特意让 location 和 age_group 列是不重复的，否则，更是随便就被锁住无法更新了。

“为了方便观察，老师回退刚才的所有操作，然后把这两列的位图索引删除了，观察 UPDATE 和 DELETE 的相关影响，如下：

```
--分别进刚才几个 SESSION 执行如下操作，完成回退
```

```
SQL> rollback;
```

```
回退已完成。
```

```
--删除 location 和 age_group 列的位图索引
```

```
SQL> drop index location_idx;
```

```
索引已删除。
```

```
SQL> drop index age_group_idx;
```

```
索引已删除。
```

脚本 5-102 暂且删除 location 和 age_group 列的位图索引，为下一试验做准备

接下来再试验看看 DELETE 语句的影响，以下例子请大家自行试验，结果我就不贴出来了，大家认真看我的说明之处，如下：

```
--位图索引之锁持有者的 DELETE 的实验
```

```
--SESSION 1（持有者）
```

```
DELETE FROM T WHERE GENDER='M' AND LOCATION=25;
```

```
--SESSION 2(其他会话) 插入带 M 的记录就立即被阻挡，以下三条语句都会被阻止
```

```
insert into t (name_id,gender,location ,age_group ,data) values (100001,'M',78, 'young','TTT');
```

```
update t set gender='M' WHERE LOCATION=25;
```

```
delete from T WHERE GENDER='M';
```

```
--以下是可以进行不受阻碍的
```

```
insert into t (name_id,gender,location ,age_group ,data) values (100001,'F',78, 'young','TTT');
```

```
delete from t where gender='F';
```

```
UPDATE T SET LOCATION=100 WHERE ROWID NOT IN ( SELECT ROWID FROM T WHERE GENDER='F' AND LOCATION=25); --update 只要不更新位图索引所在的列即可
```

脚本 5-103 请读者自行测试锁的情况

关于 UPDATE 语句，我就不多做试验了，结果大同小异，我会在后续锁的章节中再次提到如何查看锁等具体方法，这里就不做详细说明了。”

5.2.2.4 重复度低一败涂地

“大家现在明白了虽然在即席查询方面位图索引风头强劲，但是位图索引对更新却是一场灾难，因此位图索引只适用于很少更新的场合，否则将是一场噩梦！”

不过有的时候，即便很少更新，位图索引也有不适合的场合。这种情况就是，当列的取值并不是大多重复的情况下，不适合用位图索引！比如某列是姓名 ID 列，这大多不重复，绝对不适合做位图索引，而有些列比如性别、状态位、地区等列，一般来说，只会有少数几个不同的取值，这种情况下，如果该表很少更新，才适合使用位图索引。

因此位图索引的适合场景要满足两个条件：**第一，位图索引列大量重复；第二，该表极少更新。**这两个条件非常重要，切记。

大家还记得 COUNT(*) 奋勇夺冠的故事吧？”老师问。

“记得！”同学们印象非常深刻，当然不会忘记。

“大家回过头看看之前的执行计划等细节，看看老师如果在 object_id 列建立位图索引，COUNT(*) 的性能又如何，具体如下。

首先重新构造 t 表的脚本如下，执行过程就略去了：

```
drop table t purge;
create table t as select * from dba_objects;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
insert into t select * from t;
update t set object_id=rownum;
commit;
```

脚本 5-104 测试位图索引重复度前准备工作

接下来在 object_id 列建立位图索引：

```
SQL> create bitmap index idx_bit_object_id on t(object_id);
索引已创建。
SQL> select count(*) from t;
COUNT(*)
-----
3562688
```

收获，不止 Oracle

执行计划

Plan hash value: 2966233522

Id	Operation	Name	Rows	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10739 (1)	00:02:09
1	SORT AGGREGATE		1		
2	TABLE ACCESS FULL	T	3906K	10739 (1)	00:02:09

Note

- dynamic sampling used for this statement

统计信息

```
0 recursive calls
0 db block gets
48871 consistent gets
46194 physical reads
0 redo size
415 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 5-105 COUNT(*)在列重复度低时居然不使用位图索引

大家有什么发现吗？”老师问。

“老师，在 object_id 列建了位图索引后，Oracle 居然不选择这个索引而宁愿选择全表扫描！”小莲大声回答。

“说得很好，那我们看看如果强制使用这个索引，代价是多少呢？”

```
SQL> select /*+index(t,idx_bit_object_id)*/ count(*) from t;
COUNT(*)
```

3562688

执行计划

Plan hash value: 2130576087

Id	Operation	Name	Rows	Cost (%CPU)	Time
----	-----------	------	------	-------------	------

0	SELECT STATEMENT	1	12987	(1)	00:02:36
1	SORT AGGREGATE	1			
2	BITMAP CONVERSION COUNT	3906K	12987	(1)	00:02:36
3	BITMAP INDEX FULL SCAN IDX_BIT_OBJECT_ID				

Note

- dynamic sampling used for this statement

统计信息

0	recursive calls
0	db block gets
12771	consistent gets
12770	physical reads
0	redo size
415	bytes sent via SQL*Net to client
400	bytes received via SQL*Net from client
2	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
1	rows processed

脚本 5-106 强制 COUNT(*)用位图索引则性能更低

大家有什么发现吗？”老师问。

“老师，用了这个位图索引后，代价是 12987，比全表扫描的 10739 还要高，怪不得 Oracle 不用这个索引。”小莲第一个回答。

“老师，之前在 STATUS 列上建位图索引，代价才 88，逻辑读才 95，而现在代价居然是 10739，逻辑读居然是 12771，差别好大啊！”晶晶飞快地翻阅了 COUNT(*)奋勇夺冠的小节，比较了一下。

“是啊，为什么这么神奇呢？大家仔细看看老师将要做的试验，就明白了。不过这里顺便提一句，大家是否有注意到此次例子中全表扫描的逻辑读 48871 比位图索引扫描的 10739 要大得多，但是 COST 却比位图索引要小，这说明了其实逻辑读只是一个参考值，大多数情况下是可以作为调优的依据，不过 COST 显然更准确，因为 COST 除了考虑 IO，也综合考虑了 CPU 等开销。”

5.2.2.5 了解结构真相大白

“大家经历了位图索引 COUNT(*)奋勇夺冠、即席查询一骑绝尘等激动人心的场面后，正在崇尚其非凡功力时，老师给大家泼了冷水，首先是见识了位图索引中更新遭遇锁的困扰，接下来又看到在重复度低的列建位图索引时性能会一败涂地，可谓印象深刻吧。

其实我简单地说说位图索引的结构，大家就能知道为什么位图索引如此神奇了，比如如下 t

收获，不止 Oracle

表有 4 个字段，分别是 ID、NAME、SEX 和 STATUS，其中 SEX 取值仅为男或女，有时由于不知道性别，暂时为空，具体如下：

ID	NAME	SEX	STATUS
1	张一	女	毕业
2	李二	男	在读
3	王三	男	毕业
4	赵四	女	在读
5	马五		在读
6	刘六	男	毕业
7	孔七		毕业
8	叶八	男	在读
9	陈九	男	毕业
10	丁大	女	在读
.....

当我们在这个 SEX 列建位图索引时，会产生什么样的现象呢，我们先分析一下 SEX 列的取值，只有男、女和 NULL，建位图索引时，Oracle 很快知道这样的一系列索引，索引的键值只会有三个不同的取值，分别指向了表的多行，请看下面的空表，等一下我要让大家填空：

值/行	第 1 行	第 2 行	第 3 行	第 4 行	第 5 行	第 6 行	第 7 行	第 8 行	第 9 行	第 10 行
男										
女										
NULL										

请大家认真看 t 表记录，说说第一行的 SEX 列记录是男吗？”老师问。

“不是！”

“第一行的 SEX 列记录是女吗？”

“是！”

“第一行的 SEX 列记录是空吗？”

“不是！”

“很好，是就填 1，不是就填 0，刚才的表就变成如下了：

值/行	第 1 行	第 2 行	第 3 行	第 4 行	第 5 行	第 6 行	第 7 行	第 8 行	第 9 行	第 10 行
男	0									
女	1									
NULL	0									

接下来填满这些空白处，应该没有不会的吧，请一个同学上来填一下。”老师说。

小莲把手举得老高，老师请她上台来在演示板上填空，不一会儿工夫，填写如下：

值/行	第 1 行	第 2 行	第 3 行	第 4 行	第 5 行	第 6 行	第 7 行	第 8 行	第 9 行	第 10 行	……
男	0	1	1	0	0	1	0	1	1	0	……
女	1	0	0	1	0	0	0	0	0	1	……
NULL	0	0	0	0	1	0	1	0	0	0	……

“同学们，小莲同学填得对不对？”

“对！”台下的回答相当一致。

“我们再请一位同学依据之前画出 SEX 列的表格原理，把 STATUS 列建位图索引时的对应表格也画出来。”

“我来！”晶晶的手也举得老高。

不一会儿，晶晶便在演示板上画好了如下表格：

值/行	第 1 行	第 2 行	第 3 行	第 4 行	第 5 行	第 6 行	第 7 行	第 8 行	第 9 行	第 10 行	……
毕业	1	0	1	0	0	1	1	0	1	0	……
在读	0	1	0	1	1	0	0	1	0	1	……

“非常好，同学们完全明白老师的意思了，这里我再给大家强调一下，其实这个 0 和 1 就是所谓的比特值，占据的空间非常小，大小仅 1 个字节！

现在大家应该都明白关于位图索引的种种疑问了吧？”老师问。

台下一片沉默，暂时没人回答。

“没人想明白啊，再想想。”老师还是鼓励大家多思考。

“我明白了，BTREE 索引存储的是列值，而位图索引存储的是比特位值，假如位图索引所在的列只有一个取值，比如 SEX 性别列只有男，这时整个位图索引的大小大致等于行数乘以这 1 个字节，如果性别有男和女，至多也就是只有男的情况下的两倍大，所以位图索引在重复度很低时，体积非常小，所以 COUNT(*)统计非常快，老师，我猜得对吗？”晶晶问。

“说得非常好，大家都记得之前 COUNT(*)的例子，老师后来对 OBJECT_ID 这个不重复列建位图索引，之前 t 表是 300 多万条记录，意味着刚才大家画的表格不再是 2 行或者 3 行，而是 300 多万行了，大小瞬间也扩大了几百万倍，在这种情况下，在位图索引中获取条数，还有优势吗？”老师对晶晶的回答做了肯定和补充。

“现在大家都明白了为什么统计条数可以奋勇夺冠，又明白了为什么对 OBJECT_ID 建位图索引后统计条数却优势全无。接下来，大家还要分析一下，为什么即席查询可以这么迅速呢？”

“我知道了！”小莲忽然兴奋起来，“这就是以前我们学习过的与或非运算！”

同学们听得有些丈二金刚摸不着头脑，不过老师立即赞许地点了点头，笑着让小莲举例说明。

“我是看到演示板上的两个表格才明白的，大家稍等一下，我的两个表格说明快画好了。”小莲正低头画草图。

“比如我想查询 `select * from t where SEX='男' and STATUS='毕业'`，这时如果是在 SEX 和 STATUS 列分别有位图索引，情况就变成是，分别对键值为男的列和键值为毕业的列进行检索，这样 0 和 1 的与运算为 0，而 1 和 1 的与运算为 1，这样如我列的下图中，涂了阴影之处就是同时满足条件的记录了。

值/行	第 1 行	第 2 行	第 3 行	第 4 行	第 5 行	第 6 行	第 7 行	第 8 行	第 9 行	第 10 行	……
男	0	1	1	0	0	1	0	1	1	0	……
女	1	0	0	1	0	0	0	0	0	1	……
NULL	0	0	0	0	1	0	1	0	0	0	……

值/行	第 1 行	第 2 行	第 3 行	第 4 行	第 5 行	第 6 行	第 7 行	第 8 行	第 9 行	第 10 行	……
毕业	1	0	1	0	0	1	1	0	1	0	……
在读	0	1	0	1	1	0	0	1	0	1	……

分别就是第 3、6、9 等等这几行。”小莲描述得图文并茂，不免有些得意。

“回答得太完美了，现在剩下最后一个问题，为什么位图索引遭遇更新会苦不堪言呢？”老师问。

“老师，应该是唯一值越少越容易被锁吧？”曾祥不太确定。

“你说的是对的，继续。”老师笑着鼓励。

“我觉得是，比如 SEX 字段的取值只有男和女的情况，键值为男的索引条目指向了表中几乎一半以上的记录，所以更新当然影响了一半以上，我说的对吗？”在老师的鼓励下，曾祥回答得更加自信了。

“同学们真是不错，答案都是从大家的嘴里说出来的，让老师很开心，非常感谢大家的积极互动。至此位图索引的各种神秘现象背后隐藏的秘密大白于天下了。

在老师的工作中有过一个经典的案例，某生产系统忽然遭遇锁的瓶颈，插入更新频繁受阻。后来才知道，原来是其中有一张表的某字段建了位图索引导致的。这个字段为 DEAL_FLAG 字段，只有 2 个取值，0 为未处理，1 为已处理。开发人员考虑到重复度如此之高，就建了位图索引，却不料这个应用的特点如下：

- ① 插入记录到该表，DEAL_FLAG 列默认为 0，表示未处理。
- ② 处理完一定逻辑后，将 0 更新为 1，表示处理成功。

显然在这种情况下不适合建位图索引，因为这列是被频繁更新的，而且记录还是频繁插入的。最后的处理方法很简单，删除位图索引，随后系统恢复正常。

这个例子就来自一周前老师去河南出差的一次故障处理案例，现在同学们完全可以理解老师删除这个位图索引的思路吧？”老师问。

“可以！”

“至此，位图索引的章节我们暂且说到这里，我觉得位图索引好比一朵玫瑰花，美丽的背后隐藏着荆棘，不知道的人就很可能伤着手。”

5.2.3 小心函数索引步步陷阱

5.2.3.1 列运算让索引失去作用

“索引的章节马上就要结束了，现在进入最后一个环节：函数索引。在老师的相关工作经历中，遇到过不少需求以及 SQL 的写法与之有关系，因此有必要在这里和大家谈谈。

现在大家先和我一起分析一个例子，我们先利用 dba_objects 数据字典构造 t 表，并在 object_id、object_name、created 三个列上分别建了索引，完成脚本准备工作，如下：

```
SQL> drop table t purge;
表已删除。
SQL> create table t as select * from dba_objects;
表已创建。
SQL> create index idx_object_id on t(object_id);
索引已创建。
SQL> create index idx_object_name on t(object_name);
索引已创建。
SQL> create index idx_created on t(created);
索引已创建。SQL> select count(*) from t;
COUNT(*)
-----
55667
```

脚本 5-107 测函数索引前的准备

现在请大家观察这样一条 SQL 语句：select * from t where upper(object_name)='T'，显然该查询只返回一条记录，大家说说，该语句的执行计划是否用到 object_name 列上的索引？”老师问。

“可以！”台下似乎只听到一种声音。

“哦，是吗？那大家一起跟老师做个试验看看，请看如下步骤：

```
SQL> set autotrace traceonly
SQL> set linesize 1000
SQL> select * from t where upper(object_name)='T';
执行计划
```

收获，不止 Oracle

Plan hash value: 1601196873

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		9	1593	175	(1) 00:00:03
* 1	TABLE ACCESS FULL	T	9	1593	175	(1) 00:00:03

Predicate Information (identified by operation id):

1 - filter(UPPER("OBJECT_NAME")='T')

Note

- dynamic sampling used for this statement

统计信息

0 recursive calls
0 db block gets
772 consistent gets
0 physical reads
0 redo size
1198 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

脚本 5-108 对列做 UPPER 操作，无法用到索引

现在大家说说看，有用到索引吗？”老师问。

“老师，用的是全表扫描。不过 5 万多条记录仅返回 1 条，最适合用索引了，用不到是否是因为和这个 upper 写法有关，如果去掉 upper 肯定可以用到索引吧？”小莲首先回答。

“很好，小莲很善于观察，我们看看 select * from t where object_name='T' 的写法的执行计划吧，如下：

SQL> select * from t where object_name='T';

执行计划

Plan hash value: 1138138579

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
----	-----------	------	------	-------	-------------	------

```

-----
| 0 | SELECT STATEMENT | | 1 | 177 | 2 (0)| 00:00:01 |
| 1 | TABLE ACCESS BY INDEX ROWID| T | 1 | 177 | 2 (0)| 00:00:01 |
|* 2 | INDEX RANGE SCAN | IDX_OBJECT_NAME | 1 | | 1 (0)| 00:00:01 |
-----

```

Predicate Information (identified by operation id):

```

-----
2 - access("OBJECT_NAME"='T')

```

Note

```

-----
- dynamic sampling used for this statement

```

统计信息

```

-----
0 recursive calls
0 db block gets
4 consistent gets
0 physical reads
0 redo size
1198 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed

```

脚本 5-109 去掉列的 UPPER 操作后立即用索引

看来正如小莲所猜测的，是因为 upper 导致无法用索引，其实准确的说法是，对索引列做运算导致索引无法使用。

对于刚才的那个语句，如果 OBJECT_NAME 列不存在小写的字母，则 select * from t where object_name='T' 和 select * from t where upper(object_name)='T' 是完全等价的，此时如果还写这个 upper 就是多此一举而又影响性能了，我们构造的这个例子当前就是这种情况，OBJECT_NAME 的取值来自数据库新建的对象，一般都是大写的，无须在 upper 来查询了。

在很多时候，优化就是从这些点滴开始的。”

5.2.3.2 函数索引是这样应用的

“不过话又说回来，比如我们的表的 OBJECT_NAME 列的取值真的有大写又有小写，而我们查询时又需要用 upper(object_name) 来忽略大小写，这时就必须对列进行运算，在这种情况下，我们有什么好办法呢？这时函数索引就应运而生了。

建函数索引的方法很简单，和建普通索引的方法类似，区别就在于用函数运算替代列名，具

收获，不止 Oracle

体如：create index idx_upper_obj_name on t(upper(object_name))，现在我们观察一下在建了这个函数索引的情况下，是否能用到索引，效率如何，具体如下：

```
SQL> create index idx_upper_obj_name on t(upper(object_name));
索引已创建。
SQL> select * from t where upper(object_name)='T' ;
执行计划
-----
Plan hash value: 3313461867
-----
| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time |
-----
| 0 | SELECT STATEMENT | | 557 | 51801 | 113 (0) | 00:00:02 |
| 1 | TABLE ACCESS BY INDEX ROWID | T | 557 | 51801 | 113 (0) | 00:00:02 |
|* 2 | INDEX RANGE SCAN | IDX_UPPER_OBJ_NAME | 223 | | 1 (0) | 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
2 - access(UPPER("OBJECT_NAME")='T')
统计信息
-----
0 recursive calls
0 db block gets
4 consistent gets
0 physical reads
0 redo size
1198 bytes sent via SQL*Net to client
400 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

脚本 5-110 建函数索引后，对列做 UPPER 操作也可用到索引

大家观察一下，有什么发现没有？”老师问。

“老师，用到函数索引了，不过虽然逻辑读和普通索引一样是 4 个，但是代价却是 113，远大于普通索引的 2，只是比全表扫描要好，全表扫描逻辑读是 772，代价是 175。因此我认为对列建函数运算后，建函数索引的性能介于普通索引和全表扫描之间。”晶晶观察执行计划时总是又快又准确。

“说得非常好，大家看如下查询，可以分析出该表索引的类型，其中 IDX_UPPER_OBJ_NAME

索引的类型是 FUNCTION-BASED NORMAL，表示的就是函数索引。

```
SQL> select index_name, index_type from user_indexes where table_name='T';
```

INDEX_NAME	INDEX_TYPE
-----	-----
IDX_OBJECT_ID	NORMAL
IDX_OBJECT_NAME	NORMAL
IDX_CREATED	NORMAL
IDX_UPPER_OBJ_NAME	FUNCTION-BASED NORMAL

脚本 5-111 观察该函数索引的类型

其实老师之所以和大家说函数索引这个章节，除了让大家知道如何建立并使用函数索引外，还想提醒大家，在大多数情况下，对列进行函数运算的 SQL 写法都是可以转换成对列不做运算的普通写法，这才是老师认为最重要的，我会在下一小节中向大家描述这样的例子。”

5.2.3.3 避免列运算的经典案例

“其实说白了，函数索引就是因为很多情况下，SQL 写法是对列进行运算，而普通的索引没法适用于此类 SQL 语句，所以不得不建一种特殊的索引，建索引时把列名用列运算来代替，是不是这样？”老师问。

“是！”

“我们分析的结果是，在不得不使用这种列运算时，利用函数索引的性能介于普通索引和全表扫描之间，因此能用普通索引就尽量用普通索引，之前的 upper(object_id)=‘T’列的写法就可能存在 object_id=‘T’也等价的情况。现在老师要给大家展现一系列有趣的写法，大家别笑得太开心，因为这些类似的写法全部都来自身边的案例，绝非老师杜撰。

1. 经典案例 1

为了方便同学们自己可以构造分析，我们还是用刚才建立好的 t 表来举例，首先请看第一个例子，如下：

```
select * from t where object_id-10<=30;
```

大家看这个语句的写法，有什么问题？”老师问。

“老师，这个写法太有问题了吧，不就是 select * from t where object_id<=40 吗？”曾祥回答得特别大声，台下同学看到这个写法，也都大笑起来。

“大家别笑，后面的例子还有更好笑的。其实这种写法就来自你们身边的同事。大家知道 object_id 列是建过索引的，我们分别执行一下两种语句，跟踪一下性能差异，如下：

```
SQL> set autotrace traceonly
SQL> set linesize 1000
```


收获，不止 Oracle

```
SQL> select * from t where object_id-10<=30;
```

已选择 39 行。

执行计划

Plan hash value: 1601196873

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		487	86199	175 (1)	00:00:03
* 1	TABLE ACCESS FULL	T	487	86199	175 (1)	00:00:03

Predicate Information (identified by operation id):

1 - filter("OBJECT_ID"-10<=30)

Note

- dynamic sampling used for this statement

统计信息

0 recursive calls
0 db block gets
773 consistent gets
0 physical reads
0 redo size
2521 bytes sent via SQL*Net to client
422 bytes received via SQL*Net from client
4 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
39 rows processed

```
SQL> select * from t where object_id<=40;
```

已选择 39 行。

执行计划

Plan hash value: 2041828949

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		39	6903	3 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	T	39	6903	3 (0)	00:00:01
* 2	INDEX RANGE SCAN	IDX_OBJECT_ID	39		2 (0)	00:00:01

Predicate Information (identified by operation id):

```

-----
2 - access("OBJECT_ID"<=40)
Note
-----
- dynamic sampling used for this statement
统计信息
-----
0 recursive calls
0 db block gets
9 consistent gets
0 physical reads
0 redo size
2451 bytes sent via SQL*Net to client
422 bytes received via SQL*Net from client
4 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
39 rows processed

```

脚本 5-112 比较 where object_id-10<=30 和 where object_id<=40 写法的性能

同样的结果，不同的写法导致性能差异如此明显，那如果上述写法一定要写成列运算，且要用到索引，该怎么办呢？”老师问。

“建函数索引，写成 create index idx_object_id_2 on t(object_id-10)。”小莲反应很快。

“我们来试验看看，这样建是否正确以及建完后应用函数索引的情况，如下：

```
SQL> create index idx_object_id_2 on t(object_id-10);
```

索引已创建。

```
SQL> select * from t where object_id-10<=30;
```

已选择 39 行。

执行计划

```
-----
Plan hash value: 1574531187

```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		487	86199	12 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	T	487	86199	12 (0)	00:00:01
* 2	INDEX RANGE SCAN	IDX_OBJECT_ID_2	503		3 (0)	00:00:01

```
-----
Predicate Information (identified by operation id):

```

```
-----
2 - access("OBJECT_ID"-10<=30)

```

Note

- dynamic sampling used for this statement

统计信息

0 recursive calls
0 db block gets
9 consistent gets
0 physical reads
0 redo size
2451 bytes sent via SQL*Net to client
422 bytes received via SQL*Net from client
4 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
39 rows processed

脚本 5-113 你也可以让 where object_id-10<=30 用到索引

看来小莲建函数索引的方法是对的，也用到了函数索引，从性能上来说，还是介于普通索引和全表扫描之间。函数索引方式的逻辑读和普通索引相同，而 COST 是 12 要大于普通索引的 3，不过相比全表扫描的 773 个逻辑读，175 的代价值确实有了大幅度提升。

现在开始说更好笑的笑话了，该真实案例是这样的，select * from t where object_id-变量 1<=变量 2，本来我们只要改造成 select * from t where object_id<=变量 2+变量 1 即可，他们却没这么做。

开发人员做了一件让我意想不到的事情，变量 1 有 5 个固定取值，比如分别是 2，4，6，8，10。他们居然在 t 表的 object_id 列建了 5 个函数索引，建的方法大致如下：

```
create index idx_object_id_1 on t(object_id-2);  
create index idx_object_id_2 on t(object_id-4);  
create index idx_object_id_3 on t(object_id-6);  
create index idx_object_id_4 on t(object_id-8);  
create index idx_object_id_5 on t(object_id-10);
```

是不是很无语？”老师自己也忍不住笑出声来。

同学们更是放声大笑。

2. 经典案例 2

“好了，别笑了，接下来我们再看第二个例子，请看如下脚本：

```
select * from t where substr(object_name,1,4)='CLUS';
```

大家说说看，这个脚本有什么问题？”老师问。

“除非建一个 substr 相关的函数索引，否则用不上索引。”不少同学同时抢答。

“很好，那是不是就建一个函数索引呢，有其他方法改写吗？”老师问。

“我知道，这个可以改写为 select * from t where object_name like 'CLUS%'。”小莲最先发现了。

“说得太好了，这个例子也是来自身边的案例，而且也很好笑，这套系统写法类似如下：

```
select * from t where substr(object_name,1,4)='CLUS';
select * from t where substr(object_name,1,5)='CLUST';
select * from t where substr(object_name,1,6)='CLUSTE';
select * from t where substr(object_name,1,7)='CLUSTER';
```

而且他们接下来还做了这样一件事情，建了类似如下的函数索引：

```
SQL> create index idx_object_name1 on t(substr(object_name,1,4));
索引已创建。
SQL> create index idx_object_name2 on t(substr(object_name,1,5));
索引已创建。
SQL> create index idx_object_name3 on t(substr(object_name,1,6));
索引已创建。
SQL> create index idx_object_name4 on t(substr(object_name,1,7));
索引已创建。
```

是不是很无语？”这回老师终于忍住不笑了。

不过台下还是笑成了一团。

“所以在开发中，规范意识还是很重要，我就不相信，能写出 SUBSTR 函数，还具备建函数索引知识的这些开发人员，会写不出如下的等价语句：

```
select * from t where object_name like 'CLUS%';
select * from t where object_name like 'CLUST%';
select * from t where object_name like 'CLUSTE%';
select * from t where object_name like 'CLUSTER%';
```

这种语句对开发人员来说太简单了吧，只需要自问一句，你真的需要对列进行运算吗？此类笑话就可以避免了。”

3. 经典案例 3

“接下来我们看第三个例子，请看如下脚本：

```
select *
from t
where trunc(created) >= TO_DATE('2012-10-02', 'YYYY-MM-DD')
and trunc(created) <= TO_DATE('2012-10-03', 'YYYY-MM-DD');
```

收获，不止 Oracle

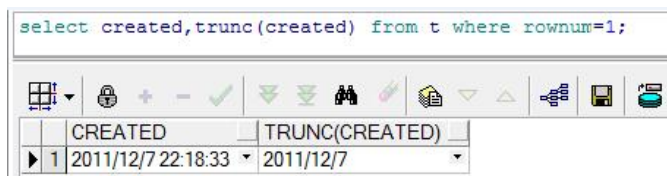
大家觉得这个写法有什么问题？”老师问。

“除非建一个 `trunc` 相关的函数索引，否则用不上索引。”依然是不少同学同时抢答。

“不错，老师要开始再问同样的问题了，该查询要用到索引，除了建一个函数索引外，有其他的方法吗？”老师继续问。

大家都开始思考了，一时没想明白。

“老师提示大家一下，请看图 5-17，这是两种写法展现的结果：



The screenshot shows a SQL query: `select created, trunc(created) from t where rownum=1;`. The result is displayed in a table with two columns: `CREATED` and `TRUNC(CREATED)`. The first row shows the value `2011/12/7 22:18:33` for `CREATED` and `2011/12/7` for `TRUNC(CREATED)`.

	CREATED	TRUNC(CREATED)
1	2011/12/7 22:18:33	2011/12/7

图 5-17 TRUNC 日期写法需谨慎

请大家再多思考一下。”梁老师始终是用一种鼓励大家思考的方式在上课。

“我明白了，可以这么写，如下：

```
select *
from t
where created >= TO_DATE('2010-10-02', 'YYYY-MM-DD')
and created < TO_DATE('2010-10-03', 'YYYY-MM-DD') + 1;
```

只要把 `<=` 改为 `<`，然后在后面加一个 1，就可以不用把 `created` 值取整，也是等价了。”小莲发现新大陆，激动得特别大声回答。

“说得太好了，这个例子也是来自身边的案例，一开始他们发现系统运行缓慢，不明白根据时间返回如此少记录的查询为什么用不到索引，后来他们查了相关资料后，在 `CREATED` 列又新增了一个函数索引，如下所示：

```
SQL> create index idx_created_2 on t(trunc(created));
索引已创建。
```

不过他们项目组要是有小莲同学，这个索引就是多余的了。”老师笑着说。

受到表扬小莲有一些不好意思了，不过老师的这三个经典案例确实是非常生动，为什么我们在写代码时，不好好地多思考一下呢？

“同学们，到此索引的章节已经全部结束了，索引课程是老师系列实用课程中最实用的部分，在今天的课程结束前，老师给大家布置一个任务，请同学们回去后认真检查自己以及自己所在项目组的 SQL 语句，看看自己能否利用索引有效改进自己的 SQL，下周我们开始新课程时请同学们先描述一下自己学以致用用的情况。”老师满怀期望地继续说。

从台下同学们的表情能感受到大家都有一种磨刀霍霍、跃跃欲试的激情。

5.3 索引让一系列最熟悉的 SQL 飞起来了

“同学们，现在开始自由分享自己这一周来的与索引有关的优化心得吧，大家记得说过的类似经历就不要重复说，所以，大家要尽量抢先发言哦。”一周时间过得很快，梁老师又在讲台上，等着分享同学们的优化实践体会。

“老师，查到我代码中有不少执行次数频繁的 COUNT(*)统计语句有性能问题，之前我从未想过这类语句和索引有啥关系，明白后我去检查代码，发现果然大多是全表扫描，在建立有效的索引后，这些语句性能提高了几十倍。

我还将所改进的 SQL 原来的执行速度和现在的执行情况做了比较，在说明增加了哪些列索引后共享给项目组，和大家一同分享，这种语句优化的适用场合真是太常见了。”

小莲第一个回答，语速因为激动而比平时快了不少。

“太好了，学以致用，真是为你高兴。”老师开心地笑了。

“还有一件事，我对部分只取少量字段返回却又频繁操作的表建了这少量字段的联合索引，让查询语句不再回表，也让一些 SQL 语句快了好多倍。”小莲又补充了一个。

“太棒了！”老师再次表扬。

“老师，我前几天帮项目组解决了一件困扰已久的性能问题。有几张很大的分区表，并且每张都有几十个分区，都建立了局部索引。我发现与这些分区表相关的语句有不少没写明分区条件，这就像您之前举例说明的一样，索引读时导致访问了几十个小索引。

后来我通过分析，将想法与项目组交流，征得同意后，我将加上分区条件后依然等价的 SQL 语句都增加了分区条件，将无法利用上分区条件的局部索引更改为全局索引。结果现在和这些分区表有关的 SQL 速度快了近百倍，大家都对我刮目相看！”

曾祥的兴奋之情溢于言表。

“挺好挺好！还有其他同学有心得吗？”曾祥兴奋的样子让老师笑得合不拢嘴。

“老师，我将我代码中一个 MAX 统计最大值的语句从原先的 400 秒提升到了 0.1 秒，就是在 MAX 取最大的这列上建上一个索引，开始项目组负责人还不同意我加这个索引，说针对所有列的统计，用索引没用，不过我将您讲述的索引原理和他交流后，他被说服了。

我加上索引的效果，让项目组所有知情人员都大吃一惊，项目经理表扬了我，同时让我把相关的案例分享成文档。这次我的优化提升效果比小莲和曾祥的百倍更明显哦，俺是千倍！”

说到这里，敬昱乐呵呵地看了眼小莲和曾祥同学，显得有些得意。

“我们项目组前几天正好遇到一个故障，系统部分菜单运行缓慢，还时常报出临时表空间不足的错误，后来被查出这个故障和我们项目组有关 SQL 有关，主要就是一些 ORDER BY 的排序。

我也尝试着一起分析探讨这些语句，发现其中有一些 SQL 语句的 ORDER BY 显然是多余的，在征得大家同意后直接去掉了这个 ORDER BY 关键字。对于不能直接去掉排序关键字部分的 SQL，我建议在这个 ORDER BY 排序列上加索引。他们开始不怎么同意，后来我把上课所学的索引与排序这部分的原理和他们分享后，他们接受我的意见并改进了。结果接下来几天，这些排序语句变快了，临时表空间不足的相关错误也消失了。”

林君看上去很沉稳，从表情上可以看出她内心的激动，不过语速却很平缓。

“非常好，这里我要特别表扬一下林君同学，她的第一意识是看看这些 ORDER BY 是否是多余的，这点做得非常好！”老师当即肯定了林君同学。

“最后到我分享了，其实刚才大家的不少优化场景我也经历过，可惜你们说过了，我不能再说了，晚出场真不好啊。”晶晶开始发言了。

大家都笑出声来了。

“我说两件挺让我得意的优化经历吧。

先从我昨天发生的一件事说起吧，印象特别深刻。我们在安徽的一个工程点的生产数据库遇到了锁等待引发严重的性能问题。由于问题是前天晚上打补丁后第二天发生的，所以在故障紧急探讨会上我提议把打补丁的语句拿出来让大家研究一下，我发现某表的某列建有位图索引，而这一列是当前状态列，显然是会频繁更改的。位图索引不能用在频繁更新的场合，这是梁老师在索引这个章节中反复强调的内容，后续将这个位图索引去掉后，系统故障立即消失。

还有一件事是前些天出账，有一张表插入速度比以前慢了近一倍！观察该表结构我发现该表居然有 9 个索引，比上个月多出了 5 个（之前的表结构有过记录），这让我非常吃惊。后来知道是这期间因某个特定的临时需求而增加了 5 个列的索引，不过现在早就不需要了，我当即建议他们把索引还原成 4 个，开始相关人员还不认可，认为索引个数和插入无关，不过在我的耐心解释下，他们还是接受了。事实证明了我是对的，删除索引后，插入的速度和上月基本一样。

这两件事我都有总结并发邮件给部门所有同事分享，我觉得能利用所学解决问题，真是一件大快人心的事啊。”说到这里，晶晶略显得有些激动。

“大家都分享差不多了，我真的很开心，我也和大家分享一个本周的和索引有关的案例吧。

老师前段时间发现某生产环境索引总数达到八千多个，而且数据库总体运行情况是索引更新方面的开销偏大，随后就开始监控这些索引。跟踪 3 个月后，昨天终于开始根据跟踪情况进行索引大整改，将 3 个月以来从未被使用过的索引全部删除了，发现平时真正被 SQL 语句使用到的索引，居然只有一千多个。现在系统的索引大幅度减少，更新比以前轻快多了，因为更新表的同时是需要更新索引的。

学习就是如此，只要善于发掘，有意识地去学习，你们就会发现，原来所学知识可以让身边最熟悉的 SQL 飞起来。”

第 6 章

经典，表的连接学以致用

6.1 表的连接之江南三剑客

表连接的相关知识汇总如图 6-1 所示。

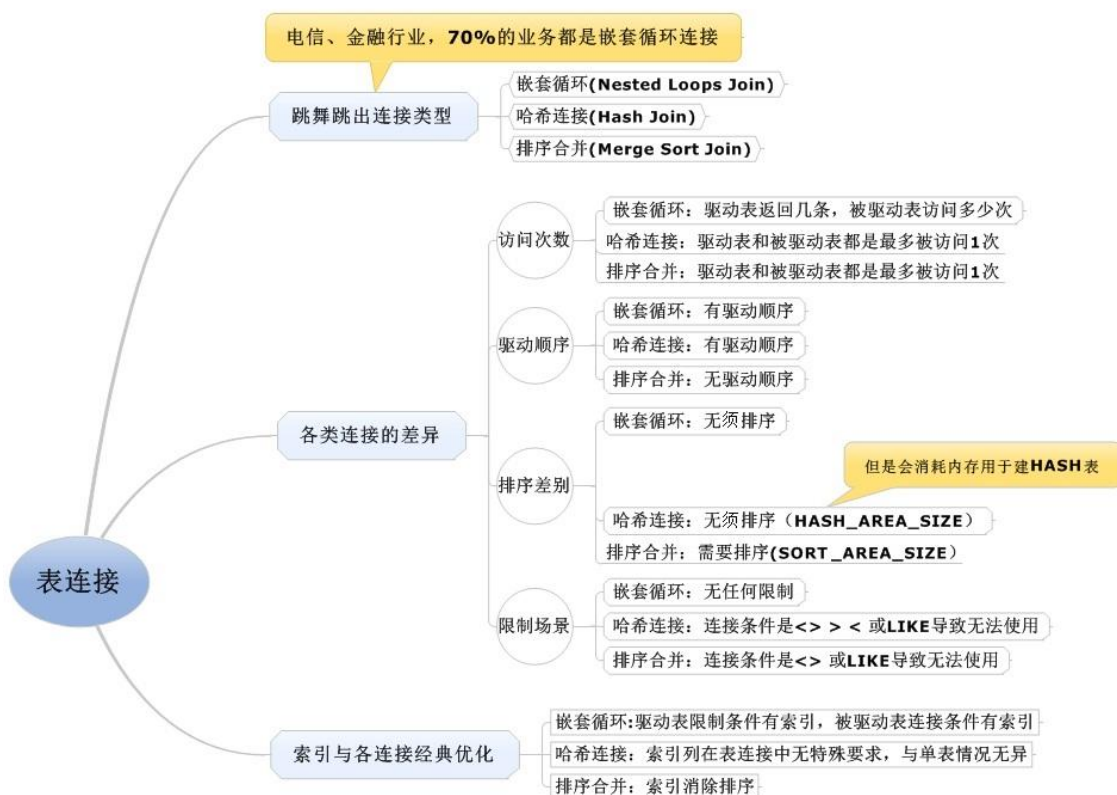


图 6-1 表连接

6.2 三大类型从小余跳舞——道来

6.2.1 跳舞也能跳出连接类型

“生活中我们随处可见，简单的背后不简单。而研究技术也不例外！”梁老师感叹道，“大家认可老师的这句话吗，谁能举例说说？”

“老师，索引结构看似简单，其实认真总结规律，可以优化一系列我们最常用的 SQL 语句，这让我觉得太震撼了，这就是简单的背后不简单。”小莲第一个说。

“体系物理结构看似简单，理解后的灵活应用可以让 SQL 语句速度从单车提升到飞船，这也是简单的背后不简单！”晶晶也起身回答。

“体系逻辑结构也是如此啊，原本我有通过官方文档了解过一些相关知识，可是从没想过有什么用，老师您让我们思考学这些知识有啥用，就是知道其实简单的背后不简单，果然受益匪浅。”曾祥也补充说。

“说得非常好，今天我们再说说看似简单其实不简单的表连接系列知识。比如 `select * from t1 t2 where t1.id=t2.id and t1.name='a'` 这个语句什么情况下最高效？”老师问。

虽然写法很简单，所有同学都了解这个 SQL 语句的含义，不过要问什么情况下最高效，还真没有想过，所以台下暂时没人回应老师。

“回答不出来没关系，这是表连接的相关知识，在上一章节讲述索引的主外键知识时，我已经简单提到过了，不过并没有展开描述。现在终于可以开始详细讲述了。在此之前，我需要先给大家说个故事，如何？”

台下又欢呼雀跃起来。

6.2.1.1 感觉怪异的嵌套循环

“小余将自己精心打扮了一番，准备参加晚上的舞会 PARTY。到了现场小余发现这是一个很特别的舞会，男孩子集中在一个房间，女孩子集中在另一个房间，而舞池却在大厅之中。

开始跳舞时先从男孩子中选出一名，然后进到女孩子所在的房间，找到高度合适的女孩子，一起到大厅中跳舞。

第二对类似如此，从男孩子所在的房间再选一名，再进女孩子所在的房间，匹配到高度适合的女孩子……

一会儿，音乐响起来，舞池中男孩女孩们在音乐声中翩翩起舞。

故事说到这里，大家有什么想法吗？”

同学们只是发笑，暂时没有人应答，小莲觉得有些怪怪的，但是说不出哪里怪。

6.2.1.2 排序合并及哈希连接

“那老师继续说了，接下来跳舞又被要求在新的规则下进行。要求男孩子在房间中进行排序，个子最矮的小余被排在了第一位。而另一间女孩子所在的房间，也做了类似的事情，最后两个房间的人都依次按顺序走到舞池大厅，依据高对高、矮对矮的原则配对，完成了舞池配对。

一会儿，音乐响起来，舞池中的男孩女孩们在一起翩翩起舞。

说到这里，我想问大家一个问题，你们觉得两次规则，哪一种更好呢，是第一种还是老师说的这第二种？”

“第二种！”台下回答得声音不仅很大，而且出奇得一致！

“为什么？”老师问。

“第一种效率明显更低啊！”林君大声说。

“看来大家观点很一致，我先不表态哪种规则更好。”老师笑着说，“现在我告诉大家，第一种方式就是数据库表连接中的嵌套循环连接（Nested Loops Join），而第二种方式就是表连接中的排序合并连接（Merge Sort Join）或者哈希连接（Hash Join），请大家记住。

按照同学们的观点，大家一定认为嵌套循环连接是比较低效的，是吧？”

“是的！”

“Oracle 既然会推出某种技术，就一定有其适用的场合，否则这个技术就不会推出来。老师向大家强调一个观点，没有最好的技术只有最合适的技术，什么时候选择什么技术，才是解决问题的关键。

因此只能说什么时候适合选择嵌套循环连接，而不是说嵌套循环连接不如排序合并或者哈希连接。

这里我们还是回到小余跳舞的故事场景中去讨论问题吧。”老师停顿了一下，“请问老师有说过男孩有多少人，女孩有多少人吗？”

“没有！”

“请问老师有说过房间中的男生有多少人参与跳舞活动，或者女生中有多少人参与跳舞活动吗？”

“没有！”

“很好，这就是问题的关键所在了，如果今天男孩中只有一名参加跳舞，那你们选择排序合并吗？”

“显然是采用嵌套循环，因为两边男女排高矮本身开销也不小，而排完后就选出小余，其他人都离去了，等同于很多人白白参与排序了。”晶晶回答。

“很好，如果男孩女孩所在的房间有很多人，而且最终是要求所有男孩和女孩都去跳舞，那你们怎么选择呢？”

“当然是先排序再合并了！”小莲不假思索地回答。

“这就是什么时候选择什么技术的问题。”老师总结说，“可是刚才我记得大家都很确定地告诉我是第二种方法更好啊。”

大家都不好意思地笑了。

“排序合并连接和哈希连接有些类似，都是基于吞吐量的操作，返回大量的甚至所有的数据时显然比嵌套循环连接要高效，但是两者还是有区别的，哈希连接不算排序，由 PGA 中的 HASH_AREA_SIZE 参数来控制，而排序合并连接则是由 PGA 中的 SORT_AREA_SIZE 参数控制的。HASH 连接使用的是 HASH 算法，比排序合并连接更高效一些，但是 HASH 连接也有诸多的限制和不便，在后续我们会说。

我在这里要告诉大家，在电信、金融等领域的数据库相关应用中，表连接总体的比例情况大致为，Nested Loops Join 占了 70%左右，而 Hash Join 占了 20%，剩下大致 10%是 Merge Sort Join。”

小莲心中暗笑，原来我们一致否认的连接方式，居然应用比例如此之高。

“大家知道为什么会是这样的比例吗？”老师继续说，“因为电信 OLTP 系统中，绝大部分的表连接查询都是返回少量的记录，好比只选小余去跳舞而非全部男生一样。”

明白了！小莲对老师举的关于小余跳舞的例子很是佩服，我怎么就没想明白呢？小莲脑子里回荡的是老师的那句话：什么时候选择什么技术。

6.2.2 各类连接访问次数差异

“现在我们开始多角度观察三种表连接的差异，首先是从表的访问次数开始研究，大家还记得之前我向大家提过一个 alter session set statistics_level=all 然后进行跟踪的方式吗？是在 5.2.1.6 节“聚合因子决定了回表查询的速度”中描述过，大家有空可以回过头去阅读一下。这种方式一个最显著的特点是，可以知道表访问的次数，现在我们开始做系列试验来研究不同类型的表连接方式的访问次数有何差异。

6.2.2.1 嵌套循环的表访问次数

首先构造 t1 和 t2 表，如下：

```
SQL> DROP TABLE t1 CASCADE CONSTRAINTS PURGE;
Table dropped
SQL> DROP TABLE t2 CASCADE CONSTRAINTS PURGE;
Table dropped
SQL> CREATE TABLE t1 (
  2   id NUMBER NOT NULL,
  3   n NUMBER,
  4   contents VARCHAR2(4000)
```

```

5 )
6 ;
Table created
SQL> CREATE TABLE t2 (
2   id NUMBER NOT NULL,
3   t1_id NUMBER NOT NULL,
4   n NUMBER,
5   contents VARCHAR2(4000)
6 )
7 ;
Table created
SQL> execute dbms_random.seed(0);
PL/SQL procedure successfully completed
SQL> INSERT INTO t1
2   SELECT rownum, rownum, dbms_random.string('a', 50)
3   FROM dual
4   CONNECT BY level <= 100
5   ORDER BY dbms_random.random;
100 rows inserted
SQL> INSERT INTO t2 SELECT rownum, rownum, rownum, dbms_random.string('b', 50) FROM dual CONNECT BY
level <= 100000
2   ORDER BY dbms_random.random;
100000 rows inserted
SQL> COMMIT;
Commit complete
SQL> select count(*) from t1;
COUNT(*)
-----
100
SQL> select count(*) from t2;
COUNT(*)
-----
100000

```

脚本 6-1 研究 Nested Loops Join 访问次数前的准备

1. 咦，T2 表为啥被访问 100 次

主要测试表连接写法如下：

```

SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id;

```

收获，不止 Oracle

接下来，我们用设置 `statistics_level=all` 的方式来观察如下表连接语句的执行计划：

```
SQL> Set linesize 1000
SQL> alter session set statistics_level=all ;
会话已更改。
SELECT /*+ leading(t1) use_nl(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id;
--略去记录结果
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID    9yfx4b0zr072k, child number 0
-----
```

```
SELECT /*+ leading(t1) use_nl(t2) */ * FROM t1, t2 WHERE t1.id = t2.t1_id
Plan hash value: 1967407726
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
	1	NESTED LOOPS		1	100	100	00:00:00.97	100K
	2	TABLE ACCESS FULL	T1	1	100	100	00:00:00.01	14
*	3	TABLE ACCESS FULL	T2	100	1	100	00:00:00.97	100K

```
-----
Predicate Information (identified by operation id):
-----
```

```
3 - filter("T1"."ID"="T2"."T1_ID")
```

```
Note
-----
```

```
PLAN_TABLE_OUTPUT
```

```
-----
- dynamic sampling used for this statement
已选择 24 行。
```

脚本 6-2 研究 Nested Loops Join，T2 表被访问 100 次

“同学们，请大家将眼睛睁大，仔细看看 **Starts** 这一列，这一列的含义是表访问的次数，请大家说说，T1 表和 T2 表分别访问多少次。”

“T1 表访问 1 次，T2 表访问 100 次！”每次抢答总是晶晶最快。

“T2 表为什么会访问 100 次呢？”老师继续问。

同学们全部无言以对。

2. 奇怪，100 次咋变成了 2 次

“看不出来，那只好再做一个试验了，大家仔细观察，在刚才的基础上，我们继续跟踪观察如下语句，差别在于增加了 t1 的条件：

```
SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n in(17, 19);
```

运行上述语句，并用 select * from table(dbms_xplan.display_cursor(null,null,'allstats last'))查询执行计划后，显示如下结果，大家注意观察：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID      b27kvf45fx0hm, child number 0
-----
```

```
SELECT /*+ leading(t1) use_nl(t2) */ * FROM t1, t2 WHERE
t1.id = t2.t1_id AND t1.n in(17, 19)
Plan hash value: 1967407726
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
1	NESTED LOOPS			1	2	2 00:00:00.02	2019
* 2	TABLE ACCESS FULL	T1		1	2	2 00:00:00.01	8
* 3	TABLE ACCESS FULL	T2		2	1	2 00:00:00.02	2011

```
-----
Predicate Information (identified by operation id):
-----
```

```
2 - filter(("T1"."N"=17 OR "T1"."N"=19))
3 - filter("T1"."ID"="T2"."T1_ID")
```

```
Note
```

```
PLAN_TABLE_OUTPUT
```

```
- dynamic sampling used for this statement
```

```
已选择 25 行。
```

脚本 6-3 换个语句，这次 T2 表被访问 2 次

现在请问 T1 表访问多少次，T2 表访问多少次？”老师再次提问。

“T1 表访问 1 次，T2 表访问 2 次！”小莲心中有些得意，她猜到老师要问这个问题了，所以早早准备，结果比晶晶回答还要快。

“那 T2 表为什么访问 2 次呢，刚才第一个例子为什么 T1 表访问 100 次呢？”老师依然问这

收获，不止 Oracle

个问题。

同学们还没想明白，依然无言以对。

“老师就是不说答案，憋死你们。”梁老师的话引来了同学们一阵哄笑。

3. 啥规律，2 次怎么变成 1 次

“那老师再做一个试验，把 SQL 语句的写法再改改，大家再观察看看，现在语句如下：

```
SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
```

继续用 select * from table(dbms_xplan.display_cursor(null,null,'allstats last'))查询执行计划，大家仔细观察结果：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
-----
SQL_ID      cg • uzv63da4y32, child number 0
-----
SELECT /*+ leading(t1) use_nl(t2) */ * FROM t1, t2 WHERE
t1.id = t2.t1_id AND t1.n = 19
Plan hash value: 1967407726
-----
| Id | Operation                | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----+-----+-----+-----+-----+-----+-----+-----+
|  1 | NESTED LOOPS              |      |       1 |       1 |       1 | 00:00:00.01 |    1014 |
|*  2 | TABLE ACCESS FULL        | T1   |       1 |       1 |       1 | 00:00:00.01 |         8 |
|*  3 | TABLE ACCESS FULL        | T2   |       1 |       1 |       1 | 00:00:00.01 |    1006 |
-----
Predicate Information (identified by operation id):
-----
   2 - filter("T1"."N"=19)
   3 - filter("T1"."ID"="T2"."T1_ID")
Note
PLAN_TABLE_OUTPUT
-----
- dynamic sampling used for this statement
已选择 25 行。
```

脚本 6—4 继续换个语句，这次 T2 表被访问 1 次

现在请……”

“T1 表访问 1 次，T2 表访问 1 次！”晶晶同学没等老师提问，就回答了，抢在了小莲的前面。老师忍不住笑起来，同学们也跟着笑了。

“那 T2 表为什么访问 1 次，之前为什么分别是 2 次和 100 次呢？”

依然没人回答，虽然小莲早知道老师问这个了，但是因为没观察明白，答不上来。

4. 不可思议，1 次又变成 0 次

“最后一次机会了，注意老师的写法，如下：

```
SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 999999999;
```

继续用 select * from table(dbms_xplan.display_cursor(null,null,'allstats last'))查询执行计划，大家仔细观察结果：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID    bpvtu4mr204u, child number 0
-----
```

```
SELECT /*+ leading(t1) use_nl(t2) */ * FROM t1, t2 WHERE
t1.id = t2.t1_id AND t1.n = 999999999
Plan hash value: 1967407726
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
1	NESTED LOOPS		1	1	0	00:00:00.01	7
* 2	TABLE ACCESS FULL	T1	1	1	0	00:00:00.01	7
* 3	TABLE ACCESS FULL	T2	0	1	0	00:00:00.01	0

```
-----
Predicate Information (identified by operation id):
-----
```

```
2 - filter("T1"."N"=999999999)
3 - filter("T1"."ID"="T2"."T1_ID")
```

Note

```
PLAN_TABLE_OUTPUT
```

```
-----
- dynamic sampling used for this statement
已选择 25 行。
```

脚本 6-5 改写到最后，T2 表居然被访问 0 次

现在请……”

“梁老师，T1 表访问 1 次，T2 表访问 0 次，而且我已经知道前面访问次数的规律了！”小莲连珠炮似的抢答又打断了老师的话，再次引来一阵笑声。

“哦，很好，那说说什么规律？”

5. NL 连接访问次数最终结论

“T1 表的查询返回多少条记录，T2 表就访问多少次。第一次 T1 表返回 100 条记录，是因为 T1 表全表就是 100 条记录，无条件查询当然就是返回 100 条；而第二次 **AND t1.n in(17,19)** 的条件让 T1 表返回 2 条记录，所以 T2 表访问 2 次；第三次 **t1.n = 19** 的条件让 T1 表的查询只返回 1 条记录，所以 T2 表访问 1 次；最后一次 **AND t1.n = 999999999** 这个条件显然是从 T1 表中查不出任何记录的，所以 T2 表访问 0 次，干脆就不访问了！”

“小莲说得大对了，也善于观察。”老师表扬了小莲接着说，“我们可以试验一下表的记录是不是如此：

---解释 T2 表为啥被访问 100 次

```
SQL> select count(*) from t1;  
COUNT(*)
```

100

---解释 T2 表为啥被访问 2 次

```
SQL> select count(*) from t1 where t1.n in (17,19);  
COUNT(*)
```

2

---解释 T2 表为啥被访问 1 次

```
SQL> select count(*) from t1 where t1.n = 19;  
COUNT(*)
```

1

---解释 T2 表为啥被访问 0 次

```
SQL> select count(*) from t1 where t1.n = 999999999;  
COUNT(*)
```

0

脚本 6-6 分析 T2 表被访问次数不同的原因

现在更加清晰了，还有不明白的吗？”老师问。

“没了！”

“此外大家要注意老师使用/*+ leading(t1) use_nl(t2)*/ 这个 HINT 的含义，其中 use_nl 表示强制用嵌套循环连接方式。Leading(t1)表示强制先访问 t1 表，也就是 t1 表作为驱动表，增加这些 HINT 提示的目的只是为了确保我们的 SQL 语句的执行计划在做嵌套循环连接，因为我们在研究这个主题，后面我们研究排序合并连接和哈希连接的访问次数时，也会用类似的 HINT 提示来固定执行计划，以方便我们进行专门课题的研究。

最后老师用一句话来总结一下规律：**在嵌套循环连接中，驱动表返回多少条记录，被驱动表就访问多少次。**”

6.2.2.2 哈希连接的表访问次数

1. 测试 T2 表仅访问 1 次

“接下来我们要讨论哈希连接是什么样的一种情况，同学们对前面关于嵌套循环连接的测试已经印象很深刻了，这里我就话不多说，直接进入测试环境，测试脚本为：

```
SELECT /*+ leading(t1) use_hash(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id;
```

开始用 select * from table(dbms_xplan.display_cursor(null,null,'allstats last'))查询执行计划，大家请注意观察 Starts 的次数，看看 T2 表被访问多少次，试验如下：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID      8waddjy8kpmcj, child number 1
-----
```

```
SELECT /*+ leading(t1) use_hash(t2) */ * FROM t1, t2 WHERE t1.id = t2.t1_id
Plan hash value: 1838229974
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
* 1	HASH JOIN		1	100	100	00:00:00.06	1019	
2	TABLE ACCESS FULL	T1	1	100	100	00:00:00.01	7	
3	TABLE ACCESS FULL	T2	1	93556	100K	00:00:00.01	1012	

```
-----
Predicate Information (identified by operation id):
-----
```

```
1 - access("T1"."ID"="T2"."T1_ID")
```

```
Note
```

```
-----
- dynamic sampling used for this statement
```

PLAN_TABLE_OUTPUT

已选择 23 行。

脚本 6-7 Hash Join 中 T2 表只会被访问 1 次或 0 次

是什么规律？”老师问。

2. HASH 连接准确结论

“T1 表虽然返回 10 条记录，T2 表仍然只访问 1 次，说明哈希连接中被驱动表的访问次数和驱动表的返回记录数无关，驱动表 T1 和被驱动表 T2 都只会被访问 1 次。”晶晶回答得很迅速。

“说的基本正确，不过更准确的说法应该是：在 HASH 连接中，驱动表和被驱动表都只会访问 0 次或者 1 次。”老师补充说。

“老师，如果驱动表返回 0 条记录，被驱动表访问 0 次这个我相信，因为之前嵌套循环是这个情况，不过您说驱动表也访问 0 次，怎么可能呢？”敬昱有些不相信。

“有疑问非常好，其实学习最重要的就是猜测、怀疑结合试验证明。我来分别构造例子来说明一下。

试验的 SQL 语句如下：

```
SELECT /*+ leading(t1) use_hash(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=999999999;
```

开始用 select * from table(dbms_xplan.display_cursor(null,null,'allstats last'))查询执行计划，大家请注意观察 Starts 的次数，看看 T2 表是否会被访问 0 次，试验如下：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
SQL_ID      fpm9vzf4vurqq, child number 0
```

```
Select /*+ leading(t1) use_hash(t2) */ * FROM t1, t2 WHERE t1.id = t2.t1_id and t1.n=999999999
Plan hash value: 1838229974
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
* 1	HASH JOIN		1	1	0	00:00:00.01	7	675K	675K	160K (0)
2	TABLE ACCESS FULL	T1	1	1	0	00:00:00.01	7			
* 3	TABLE ACCESS FULL	T2	0	93556	0	00:00:00.01	0			

Predicate Information (identified by operation id):

```

1 - access("T1"."ID"="T2"."T1_ID")
4 - access("T1"."N"=999999999)

```

Note

PLAN_TABLE_OUTPUT

- dynamic sampling used for this statement

脚本 6-8 Hash Join 中 T2 表被访问 0 次的情况

显而易见，T2 表被访问了 0 次。”老师问，“要不要继续试验一下 T1 表访问 0 次的情况？”

“要！”

“请看我设计的这个查询语句，其中 1=2 表示语句一定不成立，所以返回记录为 0：

```

SELECT /*+ leading(t1) use_hash(t2)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and 1=2;

```

开始用 select * from table(dbms_xplan.display_cursor(null,null,'allstats last'))查询执行计划，大家请注意观察 Starts 的次数，看看 T1 表是否会被访问 0 次，试验如下：

```

SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT

```

SQL_ID gk6w5q20urn4y, child number 0

```

SELECT /*+ leading(t1) use_hash(t2)*/ * FROM t1, t2 WHERE t1.id = t2.t1_id
and 1=2

```

Plan hash value: 487071653

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	OMem	1Mem	Used-Mem
* 1	FILTER		1		0	00:00:00.01			
* 2	HASH JOIN		0	100	0	00:00:00.01	732K	732K	
3	TABLE ACCESS FULL	T1	0	100	0	00:00:00.01			
4	TABLE ACCESS FULL	T2	0	93556	0	00:00:00.01			

Predicate Information (identified by operation id):

```

1 - filter(NULL IS NOT NULL)
2 - access("T1"."ID"="T2"."T1_ID")

```

PLAN_TABLE_OUTPUT

Note

- dynamic sampling used for this statement
已选择 26 行。

脚本 6-9 Hash Join 中 T1 和 T2 表都访问 0 次的情况

证明至此，大家看明白了吗？”老师问。
“明白了！”
“此外关于 NL 连接也是如此，如果刚才有 where 1=2 的写法，也必然导致 NL 连接出现驱动表和被驱动表都访问 0 次的情况。”

6.2.2.3 排序合并的表访问次数

“最后轮到排序合并连接了，我们需要测试的 SQL 写法如下：

```
SELECT /*+ ordered use_merge(t2) */ *  
FROM t1, t2  
WHERE t1.id = t2.t1_id;
```

开始用 select * from table(dbms_xplan.display_cursor(null,null,'allstats last'))查询执行计划，大家请注意观察 Starts 的次数，看看 T1 表和 T2 表的访问情况，试验如下：

SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));

PLAN_TABLE_OUTPUT

SQL_ID 7559zgttnzxhw, child number 0

SELECT /*+ ordered use_merge(t2) */ * FROM t1, t2 WHERE t1.id = t2.t1_id

Plan hash value: 412793182

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
1	MERGE JOIN		1	100	100	00:00:00.09	1012			
2	SORT JOIN		1	100	100	00:00:00.01	7	11264	11264	10204(0)
3	TABLE ACCESS FULL	T1	1	100	100	00:00:00.01	7			
* 4	SORT JOIN		100	93556	10000	00:00:00.01	1005	9266K	1184K	8236K(0)
5	TABLE ACCESS FULL	T2	1	93556	100K	00:00:00.01	1005			

Predicate Information (identified by operation id):

4 - access("T1"."ID"="T2"."T1_ID")
filter("T1"."ID"="T2"."T1_ID")

```
PLAN_TABLE_OUTPUT
```

```
Note
```

```
-----
- dynamic sampling used for this statement
已选择 26 行。
```

脚本 6-10 Merge Sort Join 访问情况和 Hash Join 一样

显然可以看出来，在访问次数上，排序合并连接和 HASH 连接是一样的，**T1 表和 T2 表都只会访问 0 次或者 1 次**。关于 0 次的试验这里就不再举例了，有兴趣的同学可以自行试验。

但是这里有一个非常重要的要点要牢记：**排序合并连接根本就没有驱动和被驱动的概念，而嵌套循环和哈希连接要考虑驱动和被驱动情况**。我将在下一讲给大家详细说明。

至此，嵌套循环连接、哈希连接、排序合并连接的访问次数规律已经全部研究清楚了，请大家好好思考一下，自行动手做试验来证实，加深印象。”

6.2.3 各类连接驱动顺序区别

“前面我们已经研究好了关于三种不同类型连接方式的访问次数差异，现在我们开始讨论它们之间关于驱动表顺序的研究，其实研究这个非常简单，我们只要观察两表的前后访问顺序对调后的性能差异即可。大家请注意看我的试验步骤。”

6.2.3.1 嵌套循环的表驱动顺序

“首先开始研究嵌套循环连接，我们准备用如下方式来观察，代码如下：

```
alter session set statistics_level=all;
SELECT /*+ leading(t1) use_nl(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));

SELECT /*+ leading(t2) use_nl(t1) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
```

我们观察执行完毕后前后两种写法的执行计划，首先观察 `/*+ leading(t1) use_nl(t2) */` 这个 T1 驱动在前的写法，如下：

收获，不止 Oracle

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID      cguzv63da4y32, child number 0
-----
```

```
SELECT /*+ leading(t1) use_nl(t2) */ * FROM t1, t2 WHERE t1.id =
t2.t1_id AND t1.n = 19
Plan hash value: 1967407726
```

```
-----
| Id | Operation                | Name | Starts | E-Rows | A-Rows |   A-Time | Buffers |
-----
|  1 | NESTED LOOPS              |      |       1 |       1 |       1 | 00:00:00.02 |    1014 |
|*  2 | TABLE ACCESS FULL | T1   |       1 |       1 |       1 | 00:00:00.01 |       8 |
|*  3 | TABLE ACCESS FULL | T2   |       1 |       1 |       1 | 00:00:00.02 |    1006 |
-----
```

Predicate Information (identified by operation id):

```
-----
  2 - filter("T1"."N"=19)
  3 - filter("T1"."ID"="T2"."T1_ID")
```

已选择 21 行。

脚本 6-11 嵌套循环连接的 T1 表先访问的情况

接下来观察/*+ leading(t2) use_nl(t1) */ 这个 T2 驱动在前的写法，如下：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID      1y8zgyfkazrt8, child number 0
-----
```

```
SELECT /*+ leading(t2) use_nl(t1) */ * FROM t1, t2 WHERE
t1.id = t2.t1_id AND t1.n = 19
Plan hash value: 4016936828
```

```
-----
| Id | Operation                | Name | Starts | E-Rows | A-Rows |   A-Time | Buffers |
-----
|  1 | NESTED LOOPS              |      |       1 |       1 |       1 | 00:00:01.36 |    701k |
|  2 | TABLE ACCESS FULL | T2   |       1 | 93556 |    100K | 00:00:00.01 |     105 |
|*  3 | TABLE ACCESS FULL | T1   |    100K |       1 |       10 | 00:00:01.30 |    700k |
-----
```

Predicate Information (identified by operation id):

```
3 - filter(("T1"."N"=19 AND "T1"."ID"="T2"."T1_ID"))
已选择 20 行。
```

脚本 6-12 Nested Loops Join 的 T2 表先访问的情况

大家观察一下，说明一下什么规律。”老师问。

“T1 表先访问的情况下 BUFFER 是 **1014**，而 T2 表先访问的情况下，BUFFER 是 **701K**，说明嵌套循环连接中驱动表的顺序非常重要，性能差异如此明显！”小莲观察老师试验时是屏住呼吸，眼都不眨一下，所以观察得分外仔细。

“说得很好！”老师表扬了一下继续问，“还有什么新发现吗？”

“老师，T1 作为驱动表被访问的情况下，T2 表只被访问了 1 次。而 T2 表作为驱动表被访问的情况下，T1 表居然被访问了 10000 次，这是因为 T1 的结果集才返回 1 条记录，而 T2 的结果集是返回 10000 条记录。”晶晶起身做了补充。

“说得非常好，因此我们得出结论，嵌套循环连接要特别注意驱动表的顺序，小的结果集先访问，大的结果集后访问，才能保证被驱动表的访问次数降到最低，从而提升性能！”老师做出了重要的总结。

6.2.3.2 哈希连接的表驱动顺序

“接下来我们开始研究哈希连接中的驱动顺序问题，我们准备用如下方式来观察，代码如下：

```
--观察 T1 先访问的情况
alter session set statistics_level=all;
SELECT /*+ leading(t1) use_hash(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));

--观察 T2 先访问的情况
SELECT /*+ leading(t2) use_hash(t1) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
```

我们观察执行完毕后前后两种写法的执行计划，首先观察 `/*+ leading(t1) use_hash(t2) */` 这个 T1 驱动在前的写法，如下：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```


收获，不止 Oracle

SQL_ID 8z4f1mxhru2kb, child number 0

SELECT /*+ leading(t1) use_hash(t2)*/ * FROM t1, t2 WHERE t1.id = t2.t1_id and t1.n=19

Plan hash value: 1838229974

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem	

* 1	HASH JOIN			1	1	100 00:00:00.04		1013	741K	741K	286K (0)
* 2	TABLE ACCESS FULL	T1		1	1	1 00:00:00.01		7			
3	TABLE ACCESS FULL	T2		1	100k	10000 00:00:00.04		1006			

Predicate Information (identified by operation id):

1 - access("T1"."ID"="T2"."T1_ID")

2 - filter("T1"."N"=19)

已选择 20 行。

脚本 6-13 HASH 连接的 T1 表先访问的情况

接下来观察/*+ leading(t2) use_hash(t1) */ 这个 T2 驱动在前的写法，如下：

SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));

PLAN_TABLE_OUTPUT

SQL_ID bqcyppnqxtrfs, child number 0

SELECT /*+ leading(t2) use_hash(t1)*/ * FROM t1, t2 WHERE t1.id = t2.t1_id and t1.n=19

Plan hash value: 2959412835

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem	

* 1	HASH JOIN			1	94	100 00:00:00.10	1013	9471K	1956K	11M (0)	
2	TABLE ACCESS FULL	T2		1	9379	10000 00:00:00.01	1005				
* 3	TABLE ACCESS FULL	T1		1	1	1 00:00:00.01	8				

Predicate Information (identified by operation id):

1 - access("T1"."ID"="T2"."T1_ID")

3 - filter("T1"."N"=19)

已选择 20 行。

脚本 6-14 Hash Join 的 T2 表先访问情况

大家观察一下，说明一下有什么发现。”老师问。

“T1 表先访问的情况下 BUFFER 是 **1013**，而 T2 表先访问的情况下，BUFFER 也是 **1013**，但是 Used-Mem 却差异明显，前者是 **286KB**，后者是 **11MB**，说明排序尺寸差异明显。再结合时间来看，前者是 **0.04 秒**，后者是 **0.1 秒**，差别也不小。

说明哈希连接中驱动表的顺序非常重要，性能也差别明显！”这次是曾祥抢先回答了，大家回答问题的积极性非常高。

6.2.3.3 排序合并的表驱动顺序

“曾祥同学回答得很好，接下来我们开始研究排序合并连接中的驱动顺序问题，我们准备用如下方式来观察，代码如下：

```
--观察 T1 先访问的情况
alter session set statistics_level=all;
SELECT /*+ leading(t1) use_merge(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));

SELECT /*+ leading(t2) use_merge(t1) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;

select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
```

我们观察执行完毕后前后两种写法的执行计划，首先观察 `/*+ leading(t1) use_merge(t2) */` 这个 T1 表先访问的情况，如下：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT

-----
SQL_ID      f2jq8unxx0218, child number 0
-----
SELECT /*+ leading(t1) use_merge(t2) */ * FROM t1, t2 WHERE t1.id = t2.t1_id and
t1.n=19
Plan hash value: 412793182
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
----	-----------	------	--------	--------	--------	--------	---------	------	------	----------

收获，不止 Oracle

	1	MERGE JOIN			1	1	100	00:00:00.07		1012				
	2	SORT JOIN			1	1	1	00:00:00.01		7		2048	2048	2048 (0)
*	3	TABLE ACCESS FULL		T1		1	1	1	00:00:00.01		7			
*	4	SORT JOIN			1	93556	100	00:00:00.07		1005		9266K	1184K	8236K (0)
	5	TABLE ACCESS FULL		T2		1	93556	100K	00:00:00.01		1005			

Predicate Information (identified by operation id):

3 - filter("T1"."N"=19)

4 - access("T1"."ID"="T2"."T1_ID")

PLAN_TABLE_OUTPUT

filter("T1"."ID"="T2"."T1_ID")

Note

- dynamic sampling used for this statement

已选择 28 行。

脚本 6-15 Merge Sort Join 的 T1 表先访问情况

接下来观察/*+ leading(t2) use_merge(t1)*/ 这个 T2 表先访问写法，如下：

SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));

PLAN_TABLE_OUTPUT

SQL_ID gy63b1zng4far, child number 0

SELECT /*+ leading(t2) use_merge(t1) */ * FROM t1, t2 WHERE t1.id = t2.t1_id and
t1.n=19

Plan hash value: 1792967693

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
1	MERGE JOIN		1	1	1	00:00:00.01	106			
2	SORT JOIN		1	93556	20	00:00:00.01	99	974K	535K	8236K (0)
3	TABLE ACCESS FULL	T2	1	93556	100K	00:00:00.01	99			
* 4	SORT JOIN		20	1	1	00:00:00.01	7	2048	2048	2048 (0)
* 5	TABLE ACCESS FULL	T1	1	1	1	00:00:00.01	7			

Predicate Information (identified by operation id):

```

4 - access("T1"."ID"="T2"."T1_ID")
      filter("T1"."ID"="T2"."T1_ID")
PLAN_TABLE_OUTPUT
-----
5 - filter("T1"."N"=19)
Note
-----
- dynamic sampling used for this statement
已选择 28 行。

```

脚本 6-16 Merge Sort Join 的 T2 表先访问情况

大家观察一下，说明一下有什么发现。”老师问。

“T1 表无论是被先访问还是后访问，效率都一样，排序尺寸一个是 2048+8236K，另一个是 8236K+2048，此外执行时间和 BUFFER 都是一样的。”还是晶晶同学第一个回答。

“说得非常好，现在我们的重要结论出来了，结论是：**嵌套循环连接和哈希连接有驱动顺序，驱动表的顺序不同将影响表连接的性能，而排序合并连接没有驱动的概念，无论哪张表在前都无妨。**”

6.2.4 各类连接排序情况分析

6.2.4.1 除嵌套循环都需排序

“现在我们再和大家探讨一个问题，嵌套循环、哈希连接、排序合并这三种表连接方式，哪种会用到排序，哪种不会用到排序，大家知道吗，需不需要老师做试验来证明一下？”老师问。

“要！”不少不注意观察的同学马上就喊出来了。

“不要证明了，之前就已经证明过了。”小莲和晶晶几乎是同时喊出来。

“老师，在嵌套循环连接的任何一次测试中，都看不到执行计划中出现 Used-Mem 相关的关键字，而哈希和排序连接的执行计划中随处可见 Used-Mem 的关键字，这不是很明显了，说明除了嵌套循环连接不需要排序外，排序合并和哈希连接都需要排序。”小莲还是抢先回答了。

6.2.4.2 排序只需取部分字段

“说得非常好，不过这里要纠正一点，哈希连接并不排序，消耗内存是用于建立 HASH 表，关于哈希连接、排序合并连接这两种连接方式，有一个很简单的优化思想大家要注意，就是不要取多余的字段参与排序，这点大家理解吗，说说为什么？”老师问。

“我知道，因为选择的字段越少，消耗内存的尺寸就越小。”晶晶立即回答。

“大家看下面两个语句，第一个语句取两表所有字段，第二个语句仅取一个字段，观察排序的尺寸：

收获，不止 Oracle

```
alter session set statistics_level=all ;
SELECT /*+ leading(t2) use_merge(t1)*/ *
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));

SELECT /*+ leading(t2) use_merge(t1)*/ t1.id
FROM t1, t2
WHERE t1.id = t2.t1_id
and t1.n=19;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
```

首先观察取全部字段的情况：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
-----
SQL_ID      gy63b1zng4far, child number 0
-----
SELECT /*+ leading(t2) use_merge(t1)*/ * FROM t1, t2 WHERE t1.id = t2.t1_id and
t1.n=19
Plan hash value: 1792967693
-----
| Id | Operation                | Name | Starts | E-Rows | A-Rows | A-Time   | Buffers | OMem | 1Mem | Used-Mem |
-----
|  1 | MERGE JOIN                |      |       1 |     1 |       1 | 00:00:00.10 |    1012 |      |      |          |
|  2 |   SORT JOIN               |      |       1 | 93556 |      20 | 00:00:00.10 |    1005 | 9266K | 1184K | 8236K (0) |
|  3 |    TABLE ACCESS FULL    | T2   |       1 | 93556 |    100K | 00:00:00.01 |    1005 |      |      |          |
|*  4 |      SORT JOIN           |      |      20 |     1 |       1 | 00:00:00.01 |       7 | 2048 | 2048 | 2048 (0) |
|*  5 |        TABLE ACCESS FULL | T1   |       1 |     1 |       1 | 00:00:00.01 |       7 |      |      |          |
-----
Predicate Information (identified by operation id):
-----
   4 - access("T1"."ID"="T2"."T1_ID")
       filter("T1"."ID"="T2"."T1_ID")
PLAN_TABLE_OUTPUT
-----
   5 - filter("T1"."N"=19)
Note
-----
   - dynamic sampling used for this statement
已选择 28 行。
```

脚本 6-17 Merge Sort Join 取所有字段的情况

再观察只取部分字段的情况：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
```

```
PLAN_TABLE_OUTPUT
```

```
-----
```

```
SQL_ID      2pkqyd3mh8q6n, child number 0
```

```
-----
```

```
SELECT /*+ leading(t2) use_merge(t1)*/ t1.id FROM t1, t2 WHERE t1.id = t2.t1_id and t1.n=19
```

```
Plan hash value: 1792967693
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	Omem	1Mem	Used-Mem	
	1	MERGE JOIN		1	94	1	00:00:00.07	1012				
	2	SORT JOIN		1	93556	20	00:00:00.07	1005	1895K	658K	1684K	(0)
	3	TABLE ACCESS FULL	T2	1	93556	100K	00:00:00.01	1005				
*	4	SORT JOIN		20	1	1	00:00:00.01	7	2048	2048	2048	(0)
*	5	TABLE ACCESS FULL	T1	1	1	1	00:00:00.01	7				

```
-----
```

```
Predicate Information (identified by operation id):
```

```
-----
```

```
4 - access("T1"."ID"="T2"."T1_ID")
```

```
      filter("T1"."ID"="T2"."T1_ID")
```

```
PLAN_TABLE_OUTPUT
```

```
-----
```

```
5 - filter("T1"."N"=19)
```

```
Note
```

```
-----
```

```
- dynamic sampling used for this statement
```

```
已选择 28 行。
```

脚本 6-18 Merge Sort Join 取部分字段的情况

大家观察一下，说一下有什么发现。”老师问。

“取部分字段排序的尺寸是 1638K+2048，显然少于取全部字段的 8236K+2048，所以取部分字段更高效！”敬昱也积极举手回答。

6.2.4.3 关于排序的经典案例

“敬昱同学回答得非常好，其实这里差异看起来还不是非常明显，实际中，如果表的记录数更大一些，字段更多一些，差别将会很明显。而且如果 PGA 的空间不够容纳排序区，将导致排序在磁盘中进行，那性能将会出现数量级的下降。

下面老师给大家说一个有关的案例，希望引起大家的重视。有一次生产系统出现相关的故障，

收获，不止 Oracle

有大致 10 条左右四表关联的 SQL 查询语句在生产中执行，这些语句涉及的表记录最大的有近百万条。由于是基于吞吐量层面的操作，要排序返回大部分记录，执行计划是走排序合并连接。最终的结果是排序的尺寸过大导致内存中无法容纳下，部分在磁盘中进行。

最终导致系统 CPU 资源被耗尽，应用缓慢，生产面临严重的性能故障。介入查询后，我发现这些语句全部是取了所有的字段参与排序合并连接，而后续相关的应用取这个结果集时又只取了部分字段，非常奇怪，向开发人员了解的结果是，用 `select *` 代码比较方便，所以这么写了，实际下一环节的应用只需要 2 个字段足矣。

最终的优化很简单，把这系列语句修改为只取部分字段，排序尺寸大大减少，并且全部由在磁盘中转换为在内存中进行，系统性能得到大幅度提升。

其实这里告诉我们一个很朴素的道理，完成同样的需求目的，做到尽量少做事甚至不做事，一定是高效的。”

6.2.5 各类连接限制场景对比

“通过前面的学习，我们了解了三种连接方式的表访问次数情况、是否有驱动表概念以及这些连接方式是否需要排序等技术细节。

现在我们继续探讨另一个主题，三种表连接方式都有哪些限制，也就是说，在什么情况下不能使用某些连接类型，这个大家知道吗？”

同学们纷纷摇头。

“那老师做系列试验来给大家演示一下，等试验完成了，大家就都明白了，在说试验之前我再次强调一下 HINT 提示，一般来说 HINT 是 Oracle 提供的用来强制走某执行计划的一个工具，比如你不想让 Oracle 的某查询走索引而要走全表扫描，你使用 `full()` 的提示就可以达成目的，反之亦是如此。但是如果你用 HINT 将导致 Oracle 的运行结果有错，或者是 Oracle 在特定场景下无法支持这个 HINT 的执行计划，那就无法如你所愿。

之前大家应该还记得 `COUNT(*)` 的优化，如果你的索引列没有定义为非空属性，你无论如何使用 `INDEX()` 的 HINT，都不可能让 Oracle 走索引的，因为索引不能存储空值，用索引来统计将得到一个错误的结果，这是无法容忍的，所以 HINT 是无法生效的。

而在表连接这个章节中，与表连接相关的三个 HINT 分别是 `use_nl`、`use_hash` 和 `use_merge`，如果我在特定的写法下，用这些 HINT 也无法达成所愿，就可以说明这是 Oracle 的一种限制，无法使用，大家听明白了吗？”

“听明白了！”

6.2.5.1 哈希连接的限制

“很好，现在我就开始做试验了，先从 HASH 连接开始进行探索，大家还记得如下语句，这

个情况下肯定是走 HASH 连接的，之前已经出现过了。

```
SELECT /*+ leading(t1) use_hash(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n = 19;
```

现在我想换一种写法，将 `t1.id = t2.t1_id` 修改为 `t1.id <> t2.t1_id`，这时 Oracle 的等值连接条件变成了不等值连接条件，这种写法 Oracle 将无法支持 HASH 连接的算法，试验如下：

```
SQL> explain plan for
  2  SELECT /*+ leading(t1) use_hash(t2) */ *
  3  FROM t1, t2
  4  WHERE t1.id <> t2.t1_id
  5  AND t1.n = 19;
已解释。
SQL> SELECT * FROM table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
```

Plan hash value: 1967407726

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		93555	363M	224 (1)	00:00:03
1	NESTED LOOPS		93555	363M	224 (1)	00:00:03
* 2	TABLE ACCESS FULL	T1	1	2028	3 (0)	00:00:01
* 3	TABLE ACCESS FULL	T2	93555	182M	221 (1)	00:00:03

脚本 6-19 Hash Join 不支持不等值连接条件

居然不根据 HINT 走 HASH 连接而是走 NL 连接，Oracle 的 HINT 不怎么听话说明这里的写法不支持 HASH 连接。

```
SQL> explain plan for
  2  SELECT /*+ leading(t1) use_hash(t2) */ *
  3  FROM t1, t2
  4  WHERE t1.id > t2.t1_id
  5  AND t1.n = 19;
已解释。
SQL> SELECT * FROM table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
```

Plan hash value: 1967407726

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		4678	18M	224 (1)	00:00:03
1	NESTED LOOPS		4678	18M	224 (1)	00:00:03
* 2	TABLE ACCESS FULL	T1	1	2028	3 (0)	00:00:01
* 3	TABLE ACCESS FULL	T2	4678	9324K	221 (1)	00:00:03

PLAN_TABLE_OUTPUT

Predicate Information (identified by operation id):

2 - filter("T1"."N">=19)

3 - filter("T1"."ID">"T2"."T1_ID")

脚本 6-20 Hash Join 不支持大于或者小于的连接条件

这里显而易见说明了 HASH 连接除了不支持 <> 连接外，大于和小于的写法也是不支持的！此外 HASH 连接也不支持 LIKE 的连接方式，具体如下：

SQL> explain plan for

```
2  SELECT /*+ leading(t1) use_hash(t2)*/ *
3  FROM t1, t2
4  WHERE t1.id like t2.t1_id
5  AND t1.n = 19;
```

已解释。

SQL> SELECT * FROM table(dbms_xplan.display);

PLAN_TABLE_OUTPUT

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		4678	18M	224 (1)	00:00:03
1	NESTED LOOPS		4678	18M	224 (1)	00:00:03
* 2	TABLE ACCESS FULL	T1	1	2028	3 (0)	00:00:01
* 3	TABLE ACCESS FULL	T2	4678	9324K	221 (1)	00:00:03

PLAN_TABLE_OUTPUT

Predicate Information (identified by operation id):

2 - filter("T1"."N">=19)

```
3 - filter(TO_CHAR("T1"."ID") LIKE TO_CHAR("T2"."T1_ID"))
```

Note

- dynamic sampling used for this statement

已选择 20 行。

脚本 6-21 Hash Join 不支持 LIKE 的连接条件

同样也可以证明，HASH 连接不支持不等值，也不支持 LIKE 等连接方式。”老师说到这里停下来，“大家都记清楚了 HASH 连接的限制没，老师总结一下：

哈希连接不支持不等值连接<>，不支持>和<的连接方式，也不支持 LIKE 的连接方式。

接下来，我们开始描述排序合并连接的相关限制。”

6.2.5.2 排序合并的限制

“大家肯定知道老师该如何测试了，方法和上面的类似，请大家仔细看试验：

```
SQL> explain plan for
```

```
2  SELECT /*+ leading(t1) use_merge(t2)*/ *
3  FROM t1, t2
4  WHERE t1.id<> t2.t1_id
5  AND t1.n = 19;
```

已解释。

```
SQL> SELECT * FROM table(dbms_xplan.display);
```

PLAN_TABLE_OUTPUT

Plan hash value: 1967407726

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		93555	363M	224 (1)	00:00:03
1	NESTED LOOPS		93555	363M	224 (1)	00:00:03
* 2	TABLE ACCESS FULL	T1	1	2028	3 (0)	00:00:01
* 3	TABLE ACCESS FULL	T2	93555	182M	221 (1)	00:00:03

PLAN_TABLE_OUTPUT

Predicate Information (identified by operation id):

2 - filter("T1"."N"=19)

3 - filter("T1"."ID"<>"T2"."T1_ID")

脚本 6-22 Merge Sort Join 不支持不等于的连接条件

收获，不止 Oracle

以上立即得出结论，排序合并连接不支持<>这样的不等值写法，接下来我们看>这样的写法是否支持：

```
SQL> explain plan for
```

```
2  SELECT /*+ leading(t1) use_merge(t2)*/ *
3  FROM t1, t2
4  WHERE t1.id>t2.t1_id
5  AND t1.n = 19;
```

已解释。

```
SQL> SELECT * FROM table(dbms_xplan.display);
```

PLAN_TABLE_OUTPUT

Plan hash value: 412793182

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time	
0	SELECT STATEMENT		4678	18M		40099	(1) 00:08:02	
1	MERGE JOIN		4678	18M		40099	(1) 00:08:02	
2	SORT JOIN		1	2028		4	(25) 00:00:01	
* 3	TABLE ACCESS FULL	T1	1	2028		3	(0) 00:00:01	
* 4	SORT JOIN		93556	182M	487M	40095	(1) 00:08:02	
5	TABLE ACCESS FULL	T2	93556	182M		221	(1) 00:00:03	

Predicate Information (identified by operation id):

3 - filter("T1"."N"=19)

4 - access(INTERNAL_FUNCTION("T1"."ID")>INTERNAL_FUNCTION("T2"."T1_ID"))
filter(INTERNAL_FUNCTION("T1"."ID")>INTERNAL_FUNCTION("T2"."T1_ID"))

Note

PLAN_TABLE_OUTPUT

- dynamic sampling used for this statement

已选择 23 行。

脚本 6-23 Merge Sort Join 支持大于或者小于的连接条件

原来排序合并连接是可以支持>这样类型的连接方式的，接下来我们观察 LIKE 方式是否支持：

```
SQL> explain plan for
```

```
2  SELECT /*+ leading(t1) use_merge(t2)*/ *
3  FROM t1, t2
4  WHERE t1.id like t2.t1_id
```

```
5 AND t1.n = 19;
```

已解释。

```
SQL> SELECT * FROM table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
```

```
-----
Plan hash value: 1967407726
-----
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		4678	18M	224 (1)	00:00:03
1	NESTED LOOPS		4678	18M	224 (1)	00:00:03
* 2	TABLE ACCESS FULL	T1	1	2028	3 (0)	00:00:01
* 3	TABLE ACCESS FULL	T2	4678	9324K	221 (1)	00:00:03

```
PLAN_TABLE_OUTPUT
```

```
-----
Predicate Information (identified by operation id):
-----
```

```
2 - filter("T1"."N">=19)
```

```
3 - filter(TO_CHAR("T1"."ID") LIKE TO_CHAR("T2"."T1_ID"))
```

Note

```
-----
- dynamic sampling used for this statement
```

已选择 20 行。

脚本 6-24 Merge Sort Join 不支持 LIKE 的连接条件

现在重要结论出来了：排序合并连接不支持<>的连接条件，也不支持 **LIKE** 的连接条件，但是比起 **HASH** 连接，支持面要广一些，支持>之类的连接条件。

那嵌套循环连接有哪些限制呢，同学们？”

6.2.5.3 嵌套循环无限制

“老师，我观察到前面两种连接方式如果由于条件限制不依据 **HINT** 走执行计划时都走了嵌套循环连接，是不是嵌套循环连接无限制啊？”小莲问。

“说得太好了，我们研究技术，最重要的一个品质就是认真观察，大胆猜测。”老师称赞了一番，“你说对了，嵌套循环就是支持所有的 **SQL** 连接条件写法，没有任何限制。”

6.3 你动手装备的表连接威震三军

“老师前面分析运行了很多关于表连接的试验脚本，我给大家 5 分钟时间，请大家仔细看完

收获，不止 Oracle

表连接相关的所有试验后，告诉老师这些试验脚本有没有什么相同的特别之处。”老师忽然问大家这个问题。

“老师，每个试验脚本都有类似 `/*+ leading(t1) use_nl(t2)*/` 之类的 HINT 提示。”5 分钟一过小莲就举手回答了，“此外，T1 和 T2 表根本就没建索引，所以所有试验脚本的执行计划都是全表扫描！”

“小莲同学善于观察，这两点说得非常好！另外老师很羞愧了，上了这么久的表连接课程，居然忘记把索引结合进来一起研究，还真得好好感谢一下小莲同学的提醒。”

大家知道老师是故意的，都笑了。

“在表连接的研究中，索引是非常重要的一部分，对提升表连接性能起到至关重要的作用。在描述索引与各类表连接的优化技巧之前，我们先来约定两个术语，一个是连接条件，一个是限制条件。

举例说明，比如如下语句：

```
select *  
  from t1, t2  
 where t1.id = t2.t1_id  
    and t1.n = 19  
    and t2.n = 5932
```

其中 `t1.id=t2.t1_id` 就是连接条件，而 `t1.n=19` 和 `t2.n=5932` 就是限制条件。

这是很重要的概念，老师将会在讲述嵌套循环连接时多次反复提及这两个名词，你们能区分清楚吗？”

“能！”

“很好，因为在电信、金融等行业应用中，NL 连接是被应用最广泛的，而且也是唯一一种没有任何限制的连接方式，所以我们先从嵌套循环连接开始说起。”

6.3.1 嵌套循环与索引

“以下语句是前面探讨嵌套循环连接时出现过的例子，大家应该印象深刻：

```
SELECT /*+ leading(t1) use_nl(t2) */ *  
FROM t1, t2  
WHERE t1.id = t2.t1_id  
AND t1.n = 19;
```

执行计划在前面试验证明环节已经罗列出来过了，在此我就不再执行，老师将执行计划直接贴出，如下：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID      cguzv63da4y32, child number 0
-----
```

```
SELECT /*+ leading(t1) use_nl(t2) */ * FROM t1, t2 WHERE t1.id =
t2.t1_id AND t1.n = 19
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
	1	NESTED LOOPS		1	1	1	00:00:00.03	1014
*	2	TABLE ACCESS FULL	T1	1	1	1	00:00:00.01	8
*	3	TABLE ACCESS FULL	T2	1	1	1	00:00:00.01	1006

脚本 6-25 Nested Loops Join 两表无索引试验

首先，为什么要使用 HINT 提示，不使用会怎么样，试验如下：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID      7xxxx7qhk4wvx, child number 0
-----
```

```
SELECT * FROM t1, t2 WHERE t1.id = t2.t1_id AND t1.n = 19
```

```
Plan hash value: 1838229974
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	Omem	1Mem	Used-Mem
*	1	HASH JOIN		1	100	100	00:00:00.03	1013	741K	741K	294K (0)
*	2	TABLE ACCESS FULL	T1	1	1	1	00:00:00.01	7			
	3	TABLE ACCESS FULL	T2	1	93556	100K	00:00:00.01	1006			

```
Predicate Information (identified by operation id):
```

```
-----
1 - access("T1"."ID"="T2"."T1_ID")
2 - filter("T1"."N"=19)
```

```
已选择 20 行。
```

脚本 6-26 两表无索引场合如果不用 HINT，一般走 Hash Join

我们发现，Oracle 认为执行计划走 HASH 连接更合理而选择了 HASH 连接，其实从执行计划来分析，两者的实际逻辑读和执行时间都差不多，应该说 Oracle 选择 HASH 连接也是一个‘艰难’的选择，如果我们用 set autotrace on 来观察，前后的 COST 的差异也相当小，因此我才用艰难两

收获，不止 Oracle

个字来形容 Oracle 选择的困难。

因为我要给大家解释 NL 连接的原理，必须展现 NL 的相关执行计划给大家看，所以老师会用 HINT 来固定执行计划，从而保证我想让 Oracle 展现什么执行计划就展现什么执行计划。

这里我想问大家一个非常重要的问题，大家觉得这个语句在没有 HINT 的情况下，是使用嵌套循环合理还是使用 HASH 连接合理？”

“老师，我觉得显然是使用 NL 连接更合理，从小余跳舞的故事来分析，如果只是小余一人参与跳舞，那第一种场景肯定是更好，也就是嵌套循环连接，可是我居然看到 Oracle 还是‘艰难’地选择了 HASH 连接，真不能理解。”晶晶同学有些疑惑地问。

“其实这是因为我们在找小余时找得辛苦，在找小余的对应舞伴时，也找得很辛苦。如果他们两人都可以迅速找到，那肯定是第一种选择方式最快了。

而索引正是快速找到某记录的利器，大家还记得我说过的索引三大特征吧，其中高度比较低正适合于此。让我们来试验一下索引的威力，看看加上索引后，能否逐步提升性能。

我们先增加 T1 表的 n 列的索引，如下：

```
SQL> CREATE INDEX t1_n ON t1 (n);
Index created
```

脚本 6-27 动手优化，对 T1 表的限制条件建索引

查看对应的执行计划如下：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID    0sv276ncn36yk, child number 0
-----
```

```
SELECT /*+ leading(t1) use_nl(t2) */ * FROM t1, t2 WHERE t1.id = t2.t1_id AND t1.n = 19
Plan hash value: 76617097
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
	1	NESTED LOOPS		1	1	1	00:00:00.02	1009
	2	TABLE ACCESS BY INDEX ROWID	T1	1	1	1	00:00:00.01	3
*	3	INDEX RANGE SCAN	T1_N	1	1	1	00:00:00.01	2
*	4	TABLE ACCESS FULL	T2	1	1	1	00:00:00.02	1006

```
-----
Predicate Information (identified by operation id):
-----
```

```
3 - access("T1"."N"=19)
4 - filter("T1"."ID"="T2"."T1_ID")
```

脚本 6-28 有了限制条件的索引，Nested Loops Join 性能略有提升

大家注意到没有，系统性能稍稍有些提升，BUFFER 从 1013 减少到 1009，提升在哪，大家知道吗？”

“因为 T1 表的扫描方式从全表扫描转化为索引读了。”小莲抢先回答。

“说得很好，不过由于 T1 表记录才 100 条，索引除了只能单块读这个劣势外，还有 TABLE ACCESS BY INDEX ROWID 的回表查询负担，所以 BUFFER 是 3，才略胜全表扫描的 BUFFER 为 7，7-3 正好等于 1013-1009，看明白没？如果 T1 表是一张超级大表，如果能通过索引读只返回一条，那两者的差距就不会只是 4 了，会有天壤之别。

接下来，我们再建一个索引，在 T2 表的 t_id 列建索引，具体如下：

```
SQL> CREATE INDEX t2_t1_id ON t2(t1_id);
Index created
```

脚本 6-29 再次动手优化，这次对 T1 表的连接条件建索引

现在我们再观察看看，BUFFER 有啥变化：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
-----
SQL_ID      11t9x8f242dm7, child number 0
-----
SELECT /*+ leading(t1) use_nl(t2) */ * FROM t1, t2 WHERE t1.id = t2.t1_id AND t1.n = 19
Plan hash value: 2669480776
-----
| Id | Operation                                | Name      | Starts | E-Rows | A-Rows | A-Time   | Buffers | Reads |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
|  1 | TABLE ACCESS BY INDEX ROWID            | T2        |       1 |       1 |       1 | 00:00:00.01 |       7 |       1 |
|  2 | NESTED LOOPS                            |           |       1 |       1 |       3 | 00:00:00.01 |       6 |       1 |
|  3 | TABLE ACCESS BY INDEX ROWID            | T1        |       1 |       1 |       1 | 00:00:00.01 |       3 |       0 |
|*  4 | INDEX RANGE SCAN                        | T1_N      |       1 |       1 |       1 | 00:00:00.01 |       2 |       0 |
|*  5 | INDEX RANGE SCAN                        | T2_T1_ID  |       1 |       1 |       1 | 00:00:00.01 |       3 |       1 |
-----
Predicate Information (identified by operation id):
-----
   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
PLAN_TABLE_OUTPUT
-----
Note
-----
   - dynamic sampling used for this statement
已选择 26 行。
```

脚本 6-30 连接条件的索引导致表连接性能有了大幅度提升

收获，不止 Oracle

天啊，居然从 1009 变为 7 了，这下我们如果不用 HINT，Oracle 应该自己会选择使用 NL 连接方式吧，测试如下：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
-----
SQL_ID      7xxxx7qhk4wvx, child number 0
-----
SELECT  * FROM t1, t2 WHERE t1.id = t2.t1_id AND t1.n = 19
Plan hash value: 2669480776
-----
| Id | Operation                                | Name      | Starts | E-Rows | A-Rows | A-Time   | Buffers | Reads |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
|  1 | TABLE ACCESS BY INDEX ROWID            | T2        |      1 |      1 |      1 | 00:00:00.01 |      7 |      1 |
|  2 |  NESTED LOOPS                            |           |      1 |      1 |      3 | 00:00:00.01 |      6 |      1 |
|  3 |    TABLE ACCESS BY INDEX ROWID          | T1        |      1 |      1 |      1 | 00:00:00.01 |      3 |      0 |
|*  4 |      INDEX RANGE SCAN                    | T1_N      |      1 |      1 |      1 | 00:00:00.01 |      2 |      0 |
|*  5 |      INDEX RANGE SCAN                    | T2_T1_ID  |      1 |      1 |      1 | 00:00:00.01 |      3 |      1 |
-----
Predicate Information (identified by operation id):
-----
   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
PLAN_TABLE_OUTPUT
-----
Note
-----
- dynamic sampling used for this statement
已选择 26 行。
```

脚本 6–31 增加了索引后 Oracle 不用 HINT，依然选择 Nested Loops Join

果然如此，Oracle 现在是无比轻松地选择了 NL 连接，而不是之前的艰难选择。由此可见，在适合 NL 连接的场景下，我们配置好精良的武器装备，就可以让 NL 连接跑得更快，在不用 HINT 的情况下，Oracle 也能做出最佳选择。

现在我来总结一下，什么情况是最合适 NL 连接的场景，什么是最精良的武器装备。

最适合 NL 连接的场景：

- ① 两表关联返回的记录不多，最佳情况是驱动表结果集仅返回 1 条或少量几条记录，而被驱动表仅匹配到 1 条或少量几条记录,这种情况即便 T1 表和 T2 表的记录奇大无比，也是非常迅速的。

② 遇到一些不等值查询导致哈希和排序合并连接被限制使用，不得不使用 NL 连接。

什么是最精良的武器装置（忘记什么是限制条件什么是连接条件的同学，请自行复习前面内容）：

- ① 驱动表的限制条件所在的列有索引。
- ② 被驱动表的连接条件所在的列有索引。

这里我要问大家三个问题，请大家回答一个问题，上述试验脚本，驱动表是哪个，被驱动表是哪个，为什么？”

“T1 表是驱动表，T2 表是被驱动表，因为执行计划的 NESTED LOOPS 范围内，T1 在前，T2 在后。”小莲回答很迅速。

“很好，第二个问题，那驱动表也就是小莲说的被先访问的 T1 表，该表的限制条件是什么，限制条件所在列是哪个列？”老师继续问。

“驱动表 T1 表的限制条件是 $t1.n=19$ ，限制条件所在的列就是 n 列。”晶晶早就猜到老师要问这个问题了。

“非常正确，那第三个问题，被驱动表也就是小莲说的被后访问的 T2 表，该表的连接条件是什么，所在的列是哪个列？”

“梁老师，对 T1 表和 T2 表来说，连接条件都是一样的吧，都是 $t1.id = t2.t1_id$ 吧？”老师话音刚落，曾祥立即就问。

“对！”老师肯定地答复了。

“那连接条件就是 $t1.id=t2.t1_id$ ，T2 表的 $t1_id$ 列就是被驱动表的连接条件。”得到老师的肯定答复后，曾祥自信地回答。

“大家都回答得这么正确，老师很高兴，大家千万别小看了老师前面说的关于索引的精良武器装备，工作中依赖这个的优化可谓数不胜数，现在请大家自行观察学习材料一分钟，体会一下老师的索引建在什么位置。”

“这小节老师讲述课程的风格和以前略有不同，我是结论说先出来了，不过原理没事先说明，大家说说，为什么嵌套循环连接要在驱动表的限制条件加索引，在被驱动表的连接条件加索引呢？”

“我知道！”善于动脑的晶晶早就思考过这个问题了，“驱动表的限制条件建索引是为了缩小扫描驱动表的时间，如果在驱动表的连接条件建索引就没什么意义了，所有列关联到另一表的所有列，等同于每条记录都要关联。而驱动表的限制条件建了索引，只快速返回 1 条或者几条，然后再传递给 T2 表的 $t2_id$ 列，一般情况下 T2 表对应 T1 表的记录返回不多，所以在 T2 表的 $t1_id$ 列建索引是有意义的。”

“说得非常好，同学们都听明白了吗，因为 NL 查询的使用范围太广了，实用性非常强，所以

收获，不止 Oracle

老师再次啰嗦一下，做一个详细的推理总结，我们就以如下 SQL 语句来推理，假如返回的结果不多，也就一条到几条。

```
SELECT *
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t1.n in (19,20)
and t2.n>=88 and t2.n<=100;
```

此时注意观察 T1 表的 `n in (19,20)` 的条件，比如导致返回 2 条记录，首先获取 `N=19` 传递给 ID 列，比如发现 ID 列值为 199，传递给 T2 表的 T1_ID 列，于是 T1_ID 也等于 199，这时等同于对 T2 表进行一个 `WHERE T2.T1_ID=199` 的查询，因为根据 NL 的适用场景，这种业务一般返回不多，1 到几条，所以用索引快速返回，这时 T2 表的其他列的索引就无意义了。

接下来 T1 表的 `N=20` 再次传递给 T1 表自己的 ID 列，比如发现值为 299，再次传递给 T2 表的 T1_ID 列，于是 T2_ID 也等于 299，这时等同于对 T2 表进行一个 `WHERE T2.T1_ID=299` 的查询，剩下的情况和 `N=19` 的情况完全类似，T2 表的 T1_ID 列非常有用，而 T2 表的其他列的索引依然无意义。

如果 `n in (19,20)` 返回很多记录，刚才的动作就要做很多遍，效率就比较低下；如果 T1 表传递给 T2 表后匹配的记录越少，T2 表的连接条件的列的索引就越能发挥作用。”

老师这么一说，小莲对表连接的索引位置一下子敏感起来了，她甚至想马上回项目组看看这些适合 NL 连接的返回少量记录的表连接查询，是否在这两个关键位置上建有索引。

6.3.2 哈希连接与索引

“说完了嵌套循环连接与索引的关系后，我们接下来要说的是哈希连接、排序合并连接与索引的关系，这两种连接类型和嵌套循环连接最大的差别在于，连接条件的索引对它们起不到传递的作用，请注意我说的传递两个字，认真听完前面嵌套循环与表连接的同学应该能明白什么意思吧。

对于哈希连接和排序合并连接来说，索引的连接条件起不到快速检索的作用，但是限制条件列如果有适合的索引可以快速检索到少量记录，还是可以提升性能的。

因此关于哈希连接与索引的关系我们就不多说了，可以理解为单表索引的设置技巧，这在之前的索引章节中已经详细叙说过了。

此外两表关联等值查询，在没有任何索引的情况下，Oracle 倾向于走哈希连接这种算法，因为哈希连接的算法本身还是比较高效先进的。哈希连接需要在 PGA 中的 `HASH_AREA_SIZE` 中完成，因此增大 `HASH_ARAE_SIZE` 也是优化哈希连接的一种有效的途径，一般在内存自动管理的情况下，我们只要加大 PGA 区大小即可。”

小莲听了心中暗自感叹，以为老师要说什么长篇大作了，原来这么快就结束了，也不举例，看来老师真是善于抓重点啊。

6.3.3 排序合并与索引

“索引对于嵌套循环连接来说非常重要，既要考虑驱动表的限制条件上的索引，又要考虑驱动表上的连接条件上的索引；而索引对于哈希连接来说，仅仅是考虑限制条件上的索引是否能用上索引，连接条件上的索引是不能发挥作用的；排序合并连接和哈希连接又有差别，排序合并连接上的连接条件虽然没有检索的作用，却有消除排序的作用，这点请大家务必注意，我们做试验来证明。

试验脚本如下：

```
alter session set statistics_level=all ;
SELECT /*+ ordered use_merge(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
--然后建索引
create index idx_t1_id on t1(id);
--有索引情况下继续观察
SELECT /*+ ordered use_merge(t2) */ *
FROM t1, t2
WHERE t1.id = t2.t1_id;
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
```

首先分析排序合并连接在连接条件未建索引情况下的执行计划：

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID   dtr69fu924znf, child number 0
-----
```

```
SELECT /*+ ordered use_merge(t2) */ * FROM t1, t2 WHERE t1.id = t2.t1_id
Plan hash value: 412793182
-----
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
	1	MERGE JOIN		1	100	100	00:00:00.11	1012			
	2	SORT JOIN		1	100	100	00:00:00.01	7	11264	11264	10240 (0)
	3	TABLE ACCESS FULL	T1	1	100	100	00:00:00.01	7			
*	4	SORT JOIN		100	93556	100	00:00:00.11	1005	9266K	1184K	8236K (0)

收获，不止 Oracle

```
| 5 | TABLE ACCESS FULL | T2 | 1 | 93556 | 100K|00:00:00.01 | 1005 | | | |
```

Predicate Information (identified by operation id):

```
4 - access("T1"."ID"="T2"."T1_ID")
    filter("T1"."ID"="T2"."T1_ID")
```

PLAN_TABLE_OUTPUT

Note

- dynamic sampling used for this statement

已选择 26 行。

脚本 6-32 连接条件未建索引，Merge Sort Join 的排序不可避免

发现排序不可避免，而且是 T1 和 T2 表分别进行排序。接下来分析在 T1 表的连接条件 ID 列上建索引，然后观察：

```
SQL> create index idx_t1_id on t1(id);
```

索引已创建。

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));
```

PLAN_TABLE_OUTPUT

SQL_ID dtr69fu924znf, child number 0

```
SELECT /*+ ordered use_merge(t2) */ * FROM t1, t2 WHERE t1.id = t2.t1_id
```

Plan hash value: 2678642687

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
1	MERGE JOIN		1	100	100	00:00:00.08	1021			
2	TABLE ACCESS BY INDEX ROWID	T1	1	100	100	00:00:00.01	16			
3	INDEX FULL SCAN	IDX_T1_ID	1	100	100	00:00:00.01	8			
* 4	SORT JOIN		100	93556	100	00:00:00.07	1005	9266K	1184K	8236K (0
5	TABLE ACCESS FULL	T2	1	93556	100K	00:00:00.01	1005			

Predicate Information (identified by operation id):

```
4 - access("T1"."ID"="T2"."T1_ID")
    filter("T1"."ID"="T2"."T1_ID")
```

PLAN_TABLE_OUTPUT

Note

- dynamic sampling used for this statement
已选择 26 行。

脚本 6-33 连接条件建索引后，Merge Sort Join 的排序减少一次

大家注意到什么变化了吗？”老师问。

“排序减少了一次！”小莲最先回答。

“很好，这里就是利用了之前索引的知识，索引本身排序，所以可以有效地避免排序合并连接中的排序。不过 Oracle 的排序合并连接本身是有缺陷的，我们在连接条件的两个列都建过索引，却只能消除一张表的排序，这点 Oracle 一直不承认，直到 Oracle 11g 的官方文档，才承认了这一点。

但是即便如此，在某些特定的场合下，我们还是可以考虑排序合并连接场合中，对连接条件列建索引，以消除一张表的排序，提升效率。

除此之外，还有一个和 HASH 连接类似的优化思路，就是增大内存排序区，避免在排序尺寸过大时在磁盘中排序。”

下篇



飞翔意识天空 ——思想与案例的分享

第 6 章 经典，表的连接学以致用

第 7 章 搞定！不靠技术靠菜刀

第 8 章 升级！靠技术改隐形刀

第 9 章 提问，也是智慧的体现

第 10 章 买鱼，居然买出方法论

第 11 章 宝典，规范让你少做事

第 7 章



搞定！不靠技术靠菜刀

本章的知识结构如图 7-1 所示。

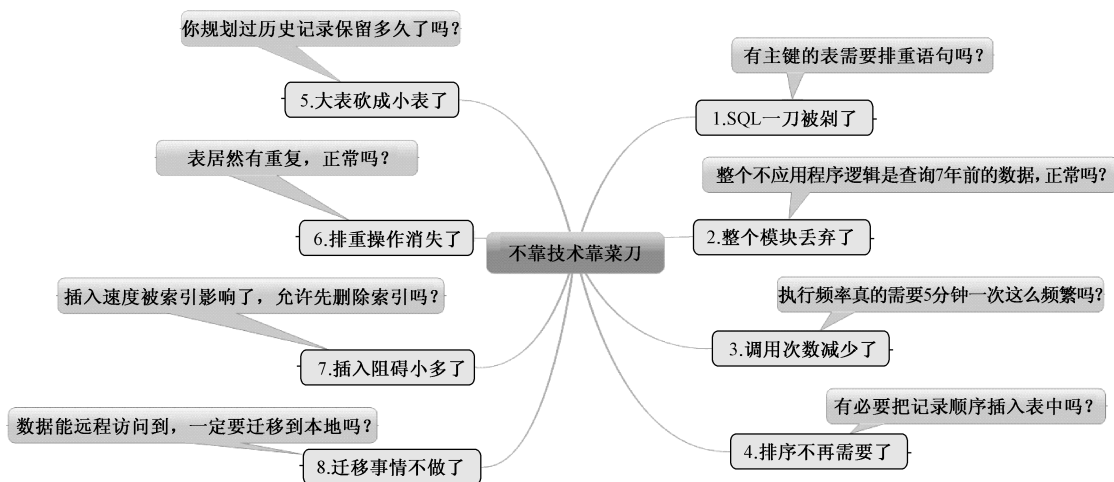


图 7-1 不靠技术靠菜刀

“同学们，现在老师给大家说一个有趣的事，我平均每解决 10 个数据库相关问题，就有 5 个和技术无关。你们信吗？”

台下有些安静，暂时没人回应，不过从不少惊异的表情来看，似乎不太相信的同学居多。

“那今天课堂上老师就开始讲自己的故事了，想听吗？”老师笑着问。

台下又是开心了一片。

“下面我说的故事其实都是一些真实的案例，主要向大家说明，在工作中的不少场景下，我们解决问题可以不依赖技术，而仅仅是靠少做事的意识来圆满完成的。这里我姑且形象地描述为不靠技术靠菜刀。”

7.1 SQL 被一刀剁了

“有一次，有个开发人员找到我，说他跟踪到自己某个存储过程中的一条 SQL 指令运行比较缓慢，让我帮忙看看，有没有什么好的优化手段。

我看到该 SQL 模块有一小段注释，大致是判断目标表 T 表中的记录中的某列 ID 列是否有重复，如果有重复保留最早插入的一条即可，这表还有另一个字段，叫 `deal_time`，记录插入的时间。

我正想把这段 SQL 取出来单独执行看看执行计划，忽然我脑海中浮现出一个想法，表记录的排重工作需要在代码中实现吗？要是这个表的 ID 列有主键，不是就已经约束了重复记录了，这段 SQL 还需要吗？带着这个疑问我看了看 T 表的表结构，看完后我大吃一惊，这个表的 ID 列居然有主键！

接下来老师会做什么事呢？”老师笑着问。

“把这段 SQL 剁了！”小莲脱口而出。

“回答得很好，表的主键本身就是用来约束保证列记录的唯一性的，这时代码中的判断变得完全多余了，后来这段代码被删除了，优化圆满完成。少做事甚至到不做事，是吧？”老师笑着说。

台下同学们也都笑了。

“大家别认为自己不会遇到这样类似的事，其实类似的例子非常多，我还能再举几个。

有一天一大早，老师就被电话给叫醒了，说新上了某个模块的补丁后，该模块运行得比以前慢多了，希望能帮忙优化。

后来发现，他们出问题的是一段类似保证工表 EMP 中的员工对应的部门，必须在部门表 DEPT 中存在的 SQL 逻辑。好比 EMP 表中的张三属于财务部门，但是部门表中不存在财务这个部门，这是不允许的。

由于开发人员代码编写不当，导致效率非常低下，已经有相关人员提出了修正的写法，性能会更好一些，正准备更改后投放应用，他们这次请我来，主要是想让我看看这个修正的写法是否可行。

其实 Oracle 在这方面已经提供了非常成熟的方案了。这分明就是一个典型的主从表中外键关联的例子。我最终的建议就是把这段逻辑删除了，在表的属性中设置这方面的约束。理由很简单，Oracle 本身实现一定比我们开发人员自己实现要来得高效，而且准确。

这一次的优化，是不是依然靠的是菜刀啊？”老师笑了。

大家听了也都乐了。

“再说一个例子吧，还有一次，有人请我帮忙调优一个存储过程，通过诊断跟踪，发现最慢的是其中一条 SQL，但是这条 SQL 的逻辑和存储过程中另外一条 SQL 的逻辑是重复的，换句话说，只要前面一条 SQL 成立，这慢的这条语句根本就无须再判断。

然后呢，同学们……”老师故意停了下来。

“把这部分 SQL 给剁了！”晶晶马上接上话来。

“对了！现实中太多的人没有尽量少做事的意识，老师这次举的三个真实的案例足以说明这一点。接下来，老师再说一个用菜刀用得最彻底的案例。”

7.2 整个模块丢弃了

“某天，我接到一个求助电话，被告知某系统运行缓慢，请求帮忙调优。到了现场后我查询到主机资源 IDLE 几乎是 0%，该主机有 12 个 CPU，居然都被耗尽，说明系统异常繁忙，仔细分析各 top 命令对应的各进程，居然发现都是同一条语句。

这下问题有些明朗了，系统遭遇的性能问题是由大量不良 SQL 同时运行引起的，奇怪的是为啥都是同一 SQL 语句，经过进一步分析发现，其实这个 SQL 是某个存储过程中的一小部分代码，该存储过程被 JOB 定时调用，每隔 5 分钟执行一次，由于该 SQL 运行缓慢，执行了 1 小时多才会结束，所以每过 5 分钟再调该存储过程时，上一次执行根本还没结束。于是这样跑啊跑的，把 12 个 CPU 都耗尽了。

这下问题更加明朗了，只要调优好这条语句，能让它在 5 分钟内执行完毕，就不会有 CPU 资源被耗尽的情况发生了。我并不着急立即动手优化这条 SQL，而是先了解一下系统是一直以来都很慢，还是忽然变慢了，这是我每次调优都会去问的一句话。

他们的答复让我有些吃惊，我被告知这个系统一直以来都是如此缓慢，忍受至今，终于受不了了。而且他们也知道是由这个 SQL 引发的，而且也知道这个 SQL 是一个定时任务，之前已经请相关人员来调优过了，也都定位到这条 SQL 并着手调优，只是效果不明显。

看来问题还有些棘手啊，因为他们居然让系统这样慢悠悠地一直跑着有些年头了。我认真看了看该 SQL 语句，该语句非常长，涉及多张表的关联，看着看着猛然间我诧异地发现，SQL 中居然看到这样一部分条件：WHERE deal_time>=to_date('2005-05-01') and deal_time<=(to_date('2005-05-31')); 这让我有些百思不得其解。现在是什么年代了，查这么多多年前某个月的数据，又不是变量，这是啥意思，这个语句现在有存在的意义吗？

经过自己多方面确认，终于弄明白了，原来这段 SQL 是 2005 年 8 月份生成的临时应急补丁，针对 5 月份的那部分数据做的某次特殊的定时逻辑处理，现在早就可以不需要了。接下来，同学们能猜到老师会怎么做吗？”

“我知道，是把这段 SQL 从过程中删除了。”敬昱大声回答。

“说得很好！不过这次不只是将该 SQL 剁了，实际情况是，我了解到该 SQL 所在的整个过程包都是完成相同目的的，都过时了。也就是说，整个 JOB 定式任务都可以不要执行了。

至此问题相当容易就解决了，停止这个定时任务，不再执行这段业务逻辑，问题立竿见影解

决了。

这是一个很有意思的案例。我没有对这段 SQL 做任何调优，至今没分析过该 SQL 的执行计划，没统计过这些 SQL 涉及的表的数据量，甚至连这段 SQL 的逻辑也没认真去看，仅仅是因为一个让人刺眼的 WHERE deal_time>=to_date('2005-05-01') and deal_time<=(to_date('2005-05-31'))条件让我对这个语句是否有存在的必要产生了怀疑。最终我对这条语句的优化效果达到了天下第一，至少是并列的。因为这条 SQL 不存在了，甚至整个与之相关的模块都不存在了，有谁做事效率能比不做更高呢？

为什么这么长时间没人想到用这种方式来解决，因为在绝大部分人的脑子里，他们认为所有的应用都是必要存在的，任务就是去改进优化这些应用，而从未考虑过是否有哪些应用程序是多余的，是绕弯路的，是可以根据业务来简化的。

因为他们缺少了少做事的意识。

这次的优化是一次典型的与调优技术无关的优化，做的仅仅是一刀剁了这个没用又耗性能的程序。看来很多时候我们可以不靠技术靠菜刀，来解决问题。”

7.3 调用次数减少了

“老师今天就是专门来描述菜刀绝招的，好听吗？”老师打趣地问。

台下无人应答，不过笑声一片。

“好了，现在请同学们再听老师说一个跟菜刀有关的案例，这次略微有些特别。

某省的监控项目上线，运行之初发现系统运行非常缓慢，基本上一天下来 CPU 的空闲率最高也只有 5%，大部分时刻 IDLE 为 0，很显然 CPU 被应用给耗尽了。

我介入调查的时候，根据 AWR 报表发现，绝大部分的 SQL 都执行得非常快，大多在 0.1 秒和 0.01 秒之间即可完成，可是这些 SQL 大多运行的次数非常之多，甚至个别语句一小时内即被执行了 1 千万次。

此时 0.01 秒的速度已经很难再提升了，但是由于 0.01 秒乘以 1 千万，总的运行时间也有 10 万秒，而系统如果只有单 CPU，一小时也不过 3600 秒而已，虽然系统的 CPU 个数达到了强劲的 64 个，依然支撑困难，所有的 CPU 都被用上了，空闲大多为 0。

因此这时的优化显然不是针对 SQL 语句的性能，而是针对执行的次数。监控的最大特点就是根据一定的周期来采样收集各个被监控系统的数据，通过收集来的数据发现各系统存在的问题。这时采样周期，或者说频率就成为当前最需要深入分析的内容了。

根据深入调查发现，该省的监控平台采样频率大多被配置为 5 分钟的周期，这是需求方需求人员最先要求的，不过显然是太频繁了，和业务规律不相符。

经过和电信需求方人员的再次确认，将不少采集模块的采集周期从 5 分钟调整为 500 分钟，

也就是推迟为半天一次采样。这个改变对系统的影响是相当大的，系统总的 SQL 调用执行次数减少到了原来的百分之一，系统自然而然就快起来了。

同学们，这个案例听起来简单还是复杂？”老师停下来问大家。

“太简单了！”台下几乎是一样的回答。

“真的吗？我再说说这个案例的特别之处，就是我介入调查时，该系统已经运行了一个多月了，这段时间系统面临着巨大的性能问题而让使用人员及维护人员相当头疼，期间也来过几波人员做过相应的优化，有的人调整了部分 SQL 的写法，有的提出了提升系统配置，也付诸了行动，早先机器的 CPU 个数仅 16 个，而现在是 64 个。

很可惜的是，系统的性能还是没能调优到让人满意的程度，其实最可惜的是，居然没人提出降低执行频率的想法，因为在大多数人的心目中，他们一直认为，SQL 执行的次数，天经地义就该是这么多。

不过我偏不这么认为，结果，我成功把执行次数给砍了不少，然后我成功了。”

7.4 排序不再需要了

“一次凌晨 2 点我被一个紧急求助电话给吵醒，被告知某系统运行缓慢且一直出现临时表空间不足的相关告警。这个告警首先就会让我们联想到是否是由于排序过多导致的，登录系统查看消耗临时表空间的语句，果然如此，占用临时表空间的 SQL 语句都是一些带有 ORDER BY 排序的语句。

该如何优化呢？首先想到的就是消除排序。那该如何消除排序呢，能否不排序呢？通过开发人员的配合，我发现这些语句居然都是一些类似如下的逻辑：

```
FOR I IN (SELECT * FROM T1 ORDER BY COL1) LOOP
    INSERT INTO T2 (COL1,COL2) VALUES (I.COL2,I.COL2);
    --略去部分代码
END LOOP;
```

这里我非常奇怪地问开发人员为什么要把 T1 表有序地取出数据再插入到 T2 表中，这有意义吗？我以为他不知道这样顺序插入不能保证 T2 表顺序地读出，结果他其实明白这个道理。但是他为什么要这么做呢？

他的回答让我大跌眼镜，他说因为这样从 T1 表中取出数据感觉更舒服。

该怎么优化呢同学们？”老师停下来问大家。

“用菜刀把排序的 ORDER BY 砍了。”曾祥的大声回答引起了台下一片笑声。

“说得很好，老师就是这么做的。

这次开发人员凌晨 1 点左右部署了一些新代码来完成一些数据抽取的相关工作，他们对程序

很有感情，想将表记录有序地读出并插入目标表中，觉得这样程序似乎会很舒服。不过实际上跑着跑着程序不怎么舒服了，因为大量的排序导致 PGA 内存区都装不下排序的尺寸，到了磁盘中去排序了。程序运行变得缓慢了。

不过真正最不舒服的还是我，凌晨两点瑟瑟发抖地从被窝里钻出来解决故障。仅仅是因为他们多做了一些根本无用的多余工作。”

7.5 大表砍成小表了

“关于排序的这个事还没说完，我将上述部分代码中多余的 ORDER BY 去掉后，排序动作大大减少了，系统也基本可以正常运行了。

不过总体来说，还是存在排序偏多的情况。因为还有一些排序的语句无法直接消除，比如有部分针对某些表依照特定的列进行排序，然后顺序取头 20 条之类的语句还是难以优化。显然需求就是如此，不能一刀把 ORDER BY 给切了，那咋办呢？

排序一时无从避免，是否能让排序的尺寸小一些呢，首先我了解了一下这些表的记录有多大，发现其中有一张 T 表特别大，记录数有 8 千多万条，很显然与这张 T 表有关的排序对当前系统的影响最大。

仔细分析该表记录情况，发现该表居然保留了 5 年前的数据（记录根本就没清除过），从业务情况分析，保留这么长的记录是相当不合理的，经过和需求方需求方人员确认，该表实际只需要保留最近三个月的记录即可。

接下来的事情非常简单，我会怎么做呢？”老师问。

“记录删除到只保留最近三个月。”小莲飞快地回答。

“很好，然后排序的问题呢？”老师继续问。

“大表变小表了，排序的尺寸也大幅度减少了，应该就解决了吧。”小莲继续回答。

“很好，从凌晨 2 点折腾到凌晨 5 点钟，问题终于解决妥当当了，老师用菜刀砍了多余的 ORDER BY 排序，又将很多无法消除排序的大表一刀砍成了小表后，终于可以安心地去睡觉了。”

7.6 排重操作消失了

“前一段时间我遇到了一件非常奇怪的事情，某项目组的项目开发人员编写的 SQL 不少都带有 distinct 排重语句，尤其是针对某些表，比如 T 表等。这让我有些不解。

这是什么原因呢，该项目组十多人写的各自不同的 SQL 都有带 distinct，难不成 T 表有重复记录？结果我一查，呆住了，还真是有重复记录。最后和项目组负责人沟通后，彻底无语了，负责人告诉我这个大量重复的 T 表是外部提供的接口表之一，他们要根据这些接口表的记录，做二

次提炼和开发，并输送返回给下一环节。因为接口表有重复记录，而该表的数据来源又非他所能控制，他也找接口表来源的相关人员反映过情况了，他们暂时无法解决重复记录问题，所以才无可奈何地在项目组中定下了针对该表的代码必须加 `distinct` 排重的规定，所以才看到如此多的语句带 `distinct`。

到此终于搞明白原因了，似乎项目经理做的也没错。

其实项目经理真是做错了，而且错得很严重！

那他要怎么做呢，同学们知道吗？”老师在这里停下来问大家。

台下一时没人应答，都还没想好。

“我的建议是，对这个接口 T 表做一个改造，比如专门针对 T 表再建一个中间表 T1 表。这个 T1 其实就是对 T 表做了排重的表。然后所有的语句的 `distinct` 写法都可以取消了，因为大家的 SQL 操作针对 T1 表而不是 T 表。

接下来这个项目组建设的系统性能一下子提升了好几倍，因为排重后，T1 表的记录比 T 表小多了，操作小表肯定比操作大表要快。而且因为 `distinct` 语句是需要排序的，只有排序后，才可以方便去重，现在排序的动作也消失了。最关键的原因是，这是大多数应用程序都需要访问的接口表，这一改进让所有相关的 SQL 全部快了，系统能不快吗？

这事最后还更加完美，就是我们协调了外部相关人员，最终彻底解决了这个 T 表的重复记录的缺陷，甚至可以允许在 T 表上建主键等约束也不会让他们程序处理出错。我们之前增加 T1 表以及相关的去重逻辑终于可以光荣退役了。

这件事情其实没有太多技术含量，但是，在我介入之前，项目组却一直没有人想过用我描述的中间表方法来改进和优化，因为技术人员中，更缺乏的是意识而不是技术。此外什么事都是需要再坚持一下，这不，最后接口表 T 表的重复记录的外部问题不也完美解决了吗？”

7.7 插入阻碍小多了

“大家都用过手机，也都很关心自己手机话费的扣费情况，现在我将要描述的案例，就和电信的计费系统有关。

一般来说，每月的月初几天计费系统要进行一次出账的工作，完成用户手机最终扣费工作，所以很多人会发现每月的头几天是无法查询到相关话费消费情况的。而这时却是电信计费系统 IT 相关支撑人员压力最大，最忙碌的时刻了。

又是一个凌晨 2 点，我又一次被求救电话叫醒，被告知计费出账时涉及的某个流程，插入账务表的动作进行得过于缓慢，每秒钟居然才插入三十几条记录，之前一般都在每秒千条以上，照这样操作下去，本月出账工作根本无法在规定的时间内完成，让我立即到现场协助支撑一下，看看有没有优化的可能。

这可是大事，我立即清醒了，匆匆到了现场，发现居然速度从每秒三十几条继续下降到十来条，相关人员都有些傻了眼了。我认真地分析了一下表的情况，发现当前账务表总记录达到了14亿多条，之前只有10亿多条，上个月记录增加很快。这也正常，和春节前后日子的特殊性有关。此外该表索引有9个之多。对比我记忆中上月该表只有5个索引的情况多了4个。

最后结合等待事件一看，基本上等待都在索引的更新上面，至此问题明白了。因为索引是需要维护的，表的记录越来越大，索引越来越多，索引的更新必须要维持索引的有序性，这是开销很大的，索引越多越大，也就会越困难。

因此我当机立断，让他们把相关索引全部失效，结果插入的速度瞬间从每秒十几条变成前所未有的每秒4000条，这个模块的工作完成得大大超出预计值。然后再把失效的索引重建，由于此时重建索引是一个批量的过程，而不是单次插入单次维护索引，速度也不慢，大致在1小时左右也全部建好。其中多出的4个索引我并没有生效，因为和有关人员探讨后，他们认为这些索引上月新建后效果不明显。

从那以后，计费的出账模式中的插入工作，就演变成了索引失效后插入记录，再生效索引的模式，直至今天。

让插入单纯一点，别边插入边维护索引。让插入阻碍小了，让插入少做事了，这就是成功的奥妙，大家听得明白吗？”看着同学们听得津津有味的样子，老师在这里停了下来。

“明白！”

“这里还有一个奥妙，就是索引重建是批量的动作，而边插入边维护索引是单次的动作，批量的性能当然好过单次的性能。好比你来回搬运砖头，是一次拿一块砖高效，还是一次推一车砖高效一样。”

7.8 迁移事情不做了

“其实老师菜刀的故事还不只是针对开发应用方面，现在我说说管理应用方面的菜刀故事。

某次有相关技术人员来找我，希望能帮忙看看为什么他们在测试环境迁移数据库一天都无法完成。我查看后发现其实是因为他们想导出的测试库的某用户有200GB那么大，而机器的配置又比较低，最关键的是，他们用的是exp/imp命令，而不是数据泵的expdp/impdp命令，前者只是针对小型数据量的导出导入的工具。

少做事的意识让我先问他们，你们为什么要从测试环境A导出这个用户到你们测试环境B中？是什么目的？

结果他们告诉我，是因为测试环境B需要读取到测试环境A下的某个用户的一些表，每次都要找测试环境A的相关人员要一些表结构和记录情况，太麻烦了。我有些吃惊地问他，那你导出你们环境后，数据不是不实时了吗？他们回答说一般变化不大可以接受，过几个月变化大了，可

以再重新导一次。

我更加吃惊了，这种事情居然还要做多次，于是我问他们，既然你们之间的网络是通的，内网的速度是惊人的千兆网，而你们又只是读取他们系统的表，为什么不建一个数据链来访问他们表，这样还需要整天迁移吗，还需要因为不够实时而反复迁移吗？

相关技术人员仿佛一下子从梦中惊醒，他冷不丁在我面前跳了起来，我怎么没想到这么简单的方法呢？

同学们，这个简单吗？”

“简单！”不少同学都听得笑出声了。

“确实简单，不过你们也不要笑，很多人也会犯类似的错误的。人的惯性思维就是，我要哪些东西，就必须取到我们本地来用，他们缺乏的不是技术，而是少做事的意识。

接下来，迁移的工作就取消了，他们在本地建了数据链来访问测试环境 A 的数据，将最经常访问的远程表建了在本地的同义词，操作起来透明得好像本地表一样，而且再也没有数据不同步的担忧了。现在大家可以去休息一下了”。

正当同学们起身要离开座位时，忽然老师叫住了大家。

“等等，刚给大家说了一个不需要迁移的案例。其实我还准备了一个需要迁移的相关案例想和大家简单分享一下，请大家稍等几分钟再休息。

某项目需要做数据迁移，需要导出的库有 2TB 大，通过 EXPDP/IMPDP 方式完成预计需要一天以上，时间太长是不允许的。正在大家束手无策时，我发现该系统某几张表连同索引就达到了近 800GB，更让人惊讶叫绝的是，该系统的回收站垃圾居然也有 800GB 这么大！

接下来进行得非常顺利，我把连同索引有 800GB 大的这些表罗列出来给需求人员开会确认，他们认为这些表大多是历史日志表，可以只导出结构，不导出记录。真是个大振奋的好消息了。我们立即清空了回收站，然后排除了这 800GB 的大表，现在我们需要导出导入的库只有 400GB 大了，真是太容易了。这个故事好听吗，同学们？”

“好听！”大家气氛非常轻松，但是可以看出来，都被老师的故事深深感染了。

“呵呵，我要纠正一下，故事前面要加上真实两个字，这些都是真实的故事。现在，大家休息一下。”

第 8 章

升级！靠技术改隐形刀

本章的知识结构如图 8-1 所示。

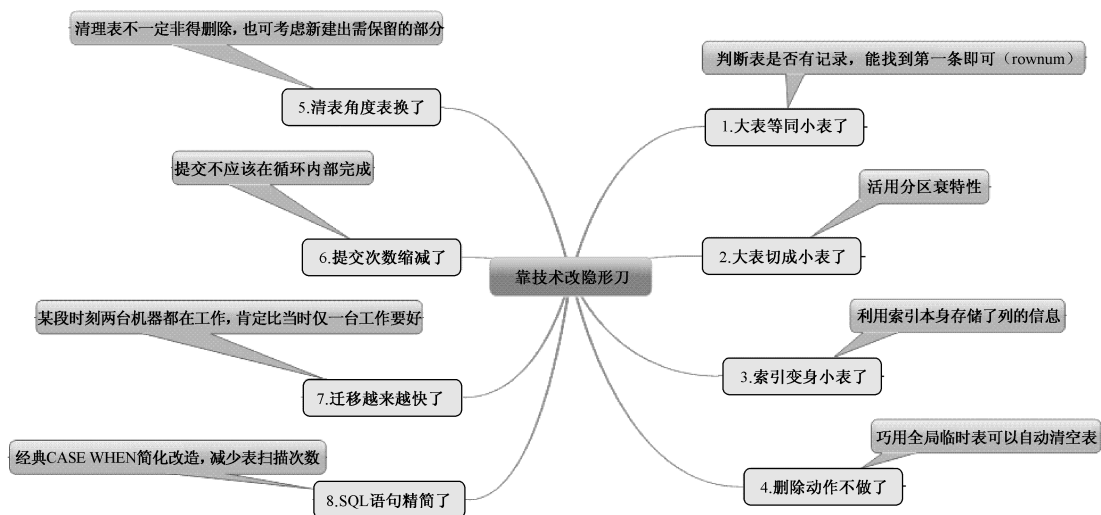


图 8-1 靠技术改隐形刀

“同学们，前面老师描述的案例是不是非常通俗易懂啊，其实这只是一种显式菜刀的描述，本质就一个字，剁！”

同学们看老师举的 8 个真实案例，其实本质就是这个不要做了，那个不要做了，然后呢？然后问题就解决了，看起来好简单啊。

不过看似简单的案例，最终还是靠我出马才得以解决，所以对缺少意识的人来说，这些简单的案例实际上并不简单。

现在我要给大家说一个稍微复杂点的案例了，但是说来说去还是说菜刀，只是现在菜刀要升级了，不是简单的这里砍砍，那里切切，这个不要了，那个不要了。而是稍微含蓄了一些，不能完全不靠技术了。通俗地说，菜刀要升级为隐形菜刀了，大家想听吗？”

不用说了，台下又是欢呼雀跃一片。

8.1 大表等同小表了

“某项目组在关键应用上线前找到我，让我审核一下其中一些比较重要模块的 PL/SQL 代码和相关数据模型，看看有没有明显不合理的地方，我答应下来了。

结果这一审核了不得了，引发了许多有趣又耐人寻味的故事，故事中的我挥舞着隐形的菜刀，在应用上线之前，砍去了许多累赘，将很多性能问题扼杀在摇篮里了。

首先映入眼帘的是类似这样的一段代码：

```
begin
select count(*) into v_cnt from t ;
if v_cnt>0
then  ...A 逻辑...
else
then  ...B 逻辑...
End;
```

这样的代码不止在一处出现，我简单搜索一下，发现让我审核的语句中，有十多处类似这样的写法，而且还都出自不同的开发人员之手。后续我把他们都找来探讨这个话题。我给他们讲了个故事后，问题就解决了。

我说比如你想到一个房间去找空箱子用，只有两种方法供你们选择，方法如下。

方法 1，把装衣服的箱子打开，一件一件地开始数有多少衣服，哇噻，好家伙，整整装了 100 件衣服！嗯，得出结论，100 件衣服大于 0，所以这个箱子不是空的，不能用。

方法 2，只需打开箱子的一小角，手伸进去，随便摸到一件衣服，嗯，这个箱子不是空的，不能用。

你们是选方法 1 还是方法 2 呢？

他们以为我在开玩笑，都笑着说肯定是方法 2。

于是我把这些代码指出来给他们看，问他们，你们都会选择用方法 2，那你们判断表 T 是否有记录时，为啥一定要通过记录总数是否大于 0 来判断呢，如果有一条记录，不就表示有记录了吗？

他们顿时明白了我讲故事的意思了，其中有一个叫起来了，说可以加上 rownum=1 的条件，然后通过记录返回是否为 1 来判断，就可以不用搜索整个表的记录了。

他们明白意思了，你们明白了吗？”老师问大家。

“明白！”

“很好，现在的语句更改为如下：

```
begin
select count(*) into v_cnt from t where rownum=1;
```

```

if v_cnt=1
then  ...A 逻辑...
else
then  ...B 逻辑...
End;

```

这次我没有非常直截了当地砍掉这个砍掉那个，而是含蓄地用加上 `rownum=1` 的写法从读全表记录到只读表的第一行记录，隐形的菜刀砍去了大量多余的数据访问路径。

这里涉及了一点点与数据库相关的但是特别简单的知识，就是 `rownum` 伪列的使用，这是一个不等价的改写，但是理解了真正需求后，我们居然发现是等价的。是什么促使我们去这么思考呢？不用说，当然是少做事的意识！

事实证明这次用隐形菜刀砍去的多余访问路径是非常有必要的，因为代码中的这些涉及统计条数的表有部分是记录达到千万级别的大表。相关开发人员还一直头疼该如何解决这个统计大表记录数的问题。

现在开发人员不必担忧这个问题了，因为不管该表变得多大，我们始终只访问该表的第一条而已。现在对这个模块的应用来说，等同与把一张千万大表神奇地变成了只有一条记录的小表了。这把隐形的菜刀砍记录砍得真是大快人心啊。”

8.2 大表切成小表了

“在这个项目组代码的审查工作中，我发现存储过程中有一个普通的代码，大致如下：`Select * from t where 列1=xxx and 列2=xxx ……`，这语句本身没啥特别的，但是该语句之前的一条注释引起了我的注意，这个注释写明了该 SQL 语句获取最近的交易记录情况。

我非常奇怪的是既然注释说明是反应最近交易情况，那代码怎么看不到有任何反映‘最近’两字的时间范围条件呢？带着这个疑问我询问了开发人员。他告诉我因为这几个条件本身就决定了是最近一个月的记录，不可能在更早以前，所以不必加时间范围字段，加了还挺麻烦的，还需要传入时间变量的参数。

真是个好孩子，还想着能不用写上条件代码，开发中尽量少做事，是这样吗同学们？”老师停下来问大家。

“应该是。”小莲有些不确定地回应了老师。

“事实上后续我让他们补上了时间条件，按他注释说的，这些条件只可能导致查询到最近一个月的数据，那就把最近一个月的条件加上吧。通过数据模型看出该表是一张按月做了分区的分区表，假如系统运行了一年，该表就会有 12 个分区装满数据，既然你明确语句只会查最近一个月的记录，那就请你把时间条件加上吧。加上当前月的条件，你的访问将会从遍历 12 个分区变为遍历 1 个分区，等同于表的访问路径缩减了 90% 以上，大表切割成小表，性能也将会大幅度提升。

通过测试环境模拟真实数据量情况，果然发现新旧两种写法性能差别在 10 倍以上。对于此类查询来说，带上分区条件后，等同于大表变成了缩减十分之一的小表了。

同学们，隐形菜刀比起之前说的直截了当的菜刀更具有一些技术含量，它们都是少做事的意识，都需要去理解业务和把握需求。但是隐形菜刀除了意识，还需要技能，比如需要你了解 rownum 伪列，需要你学习过分区表知识等。此外手法也不是除了删就是剁的那样露骨方式，而是相对比较巧妙一点，比如这两个案例中，大表真的变成小表了吗？非也，只是对于某些应用来说，你只需要访问非常少的一部分记录，等同于大表变小表了，等同于你用菜刀将大表砍去大量多余数据。

接下来继续听老师讲述隐形菜刀的系列故事。”

8.3 索引变身小表了

“接下来继续审核系列 SQL 语句，我发现有一系列为数不少的 SUM(某列)的统计语句，比如 SELECT SUM(COL1) from t 之类的 SQL 语句，我特意检查了 T 表的表结构，发现表 T 有 30 多个列，而 COL1 列属性为非空，且未建索引。

之前给大家上过索引的课程，大家应该有印象，索引本身可以存储列值，所以如果此时在 COL1 列建索引，就可以通过索引直接得出 SUM(COL1)的记录之和，而无须从表中获取 COL1 的记录开始汇总计算。

那差别是什么呢？一个表有 40 多个列，而只对其中一个列建索引，这个索引的大小必然比表小得多，在同样可以得出 COL1 记录汇总值的情况下，显然用索引更好。

这是因为……”老师故意停下来，等同学们回答。

“这是因为索引的大小要比表小得多，这时索引就等同于一个小表。访问大表变成访问小表，当然快要得多！”晶晶飞快地接下了老师的话。

同学们纷纷点头，索引这个章节是同学们公认收获最大的一个章节，什么样类型的 SQL 建什么样的索引，他们早已得心应手了。

“我知道你们都懂，为什么老师举这个例子，其实就是在说明，其实想让大表变小表，想要访问路径大大缩减，方法是多种多样的。从之前的 rownum=1 到加上分区条件，让查询只落在特定的分区，大家应该印象也很深刻了吧，当然前一章节描述的直接手法虽然露骨了一些，但很多时候，也是一个利器！”

8.4 删除动作不做了

“前段时间我们部门的某产品在湖北工程点上线运营，运行一段时间后，发现系统存在日志产生过多、日志切换太频繁的情况，观察主机的运行情况，发现系统的 IO 相当繁忙。

跟踪系统的 AWR 报表，发现了一个惊人的情况，一条类似 `delete from t where xxx` 的简单删除语句，一天下来居然被执行了 500 万次，产生了大量的日志，看来产生大量日志首先就要‘归功于’这条执行次数如此频繁的删除语句了。

后来和相关模块的开发人员确认，发现这个 T 表其实是一个临时处理的中间表，插入一些数据，进行合并运算后，将数据从中间表 T 表转移到目标表中，实现的是这样的逻辑。听了他的介绍，我立即想到了全局临时表，因为全局临时表最大的特点就是，处理完该表后，退出 SESSION，该表记录自动删除了，不需要我们有删除的动作，自然也不会产生日志。我问他是否可以考虑使用全局临时表时，他的回答让我小小的吃了一惊，你们知道他回答什么吗？”老师故意停下来问大家。

同学们摇摇头。

“他问我，什么叫全局临时表？”

台下有人忍不住笑出声来。

“不过不知道这个全局临时表的特性也算正常，于是我就花了一点时间给他描述了一番，结果他惊叫起来，哇塞，还有这么好的特性啊，我正需要这个特性！”

我和他一起更仔细地分析了他的业务逻辑，发现他的这个应用真是最适合使用全局临时表的应用，现在他对 T 表的操作由于并发较多，由于代码的部分漏洞，还经常产生死锁的情况。而全局临时表另一个显著的特点是不同的 SESSION 看到 T 表不同的记录，无论何时都不会互相锁。

经过代码的改写和表的重新设计，系统中原先耗去大量日志的删除语句从每天执行近 500 万次变成了 0 次，消失了。

这次动作很像前一章中的赤裸裸的菜刀，直接把删除语句在代码中去除了，不过这次可不是因为删除动作是多余的，而是因为全局临时表有这个退出 SESSION 自动删除记录的动作，而且还不产生日志。

这次优化缓解了生产系统 IO 繁忙、日志产生过多的问题，效果非常显著！为什么每次总是要我出马才能解决，为什么描述起如何解决听起来又总是那么简单？归根结底还是因为太多的人缺乏尽量少做事的意识。

这一次，我挥舞着手中的菜刀，哦，准确地说是稍微含蓄一点的隐形菜刀，砍去了删除表的大量动作，然后，少做事了，然后我成功了。”

8.5 清表角度变换了

“大家应该还记得前一章中我有一段关于排重 DISTINCT 相关的案例描述吧，这里我来说说这个案例的续集。

前面说到后来通过协调，把源头接口表的重复记录问题直接搞定了，这时接口表对应的程序

端再也不会送重复记录到接口表中了，只是现在我们要做的是，将接口表的重复记录删除。这看起来是一件很简单的事情，没想到，连续操作了两个晚上，都没能成功。

后来接口表模块的相关技术人员找到我，想让我帮忙删除一下该表的重复记录，我很奇怪地问他们，这个找到重复记录并删除的操作，很难吗？

他们有些无可奈何地告诉我，现在该表记录达到四千多万条，而重复的记录居然达到三千八百多万条，两个晚上都是从凌晨 2 点停应用，然后着手开始删除重复记录，然而直到凌晨 4 点还未能删除完毕。而 4 点后应用程序根据业务需要必须启动，这时删除就会因为锁而失败。两个晚上都是如此，所以想请我帮忙。

原来是这个原因，我明白了。转念一想，他们真需要这样删除记录吗，你们谁能想出更好的办法呢？”老师忽然停下来问大家。

台下沉默了一小会儿，忽然有个声音大声传来：“老师，四千万条记录删除三千八百多万条重复记录，就只保留一百多万条不重复记录，那为什么不单独把这一百万条不重复记录取出来建成表，然后把原表删了，用建出来的新表，不是要快得多吗？”小莲有些想明白了，随机脱口而出。

“非常好，要是当时你在现场，你就能帮他们解决问题了。”老师笑着肯定了小莲的回答。

“确实是，不重复记录只占了重复记录的四十分之一，很显然我们把这部分记录 CREATE 成一张表，比如 T_NEW，然后把原先的 T 表 drop 删除了，然后将 T_NEW 重命名回 T 表，最后补上一些表对应的索引，问题就解决了。

同样是实现清理重复记录的目的，这个方法显然少做事。最终的效果非常振奋人心，用这个方法来清理数据，仅仅 15 分钟就完成了，他们懊恼了两个晚上，终于喜上眉梢了。同学们，这个案例难度大吗？”

“不大！”

“确实难度不大，甚至可以说很简单。可惜，总是有那么多人缺少意识。因为缺少意识，相关操作人员连续两次凌晨 2 点到 4 点辛苦的工作白白付出了。”

8.6 提交次数缩减了

“这次我说的案例，还是和刚才提到的湖北工程点的案例有关。在用全局临时表解决了日志产生过多的问题后系统平稳运行了一段时间，忽然又出现问题了，有人反映系统运行有些缓慢。

介入调查后观察 AWR 报表，发现等待事件中与 commit 事件有关的 log file sync 等待非常显著，一周时间内等待了 3918701 次近 400 万秒，很显然数据库应用存在单次提交过频繁，未有效批量提交的情况。具体如图 8-2 所示。

Top 5 Timed Events

Event	Waits	Time(s)	Avg Wait (ms)	% Total Call Time	Wait Class
CPU time		5,005,331		29.7	
log file sync	335,927,693	3,918,701	12	23.3	Commit
SQL*Net message from dblink	23,467,420	1,764,194	75	10.5	Network
log file switch (checkpoint incomplete)	9,004,575	1,115,677	124	6.6	Configuration
RMAN backup & recovery I/O	24,841,428	529,047	21	3.1	System I/O

图 8-2 AWR TOP5

通过如下查询，发现某模块有一个更新语句非常频繁，查出某 SESSION 登录仅仅不过 6 小时就提交了二百多万次，因为具体定位问题的方法具有一定的实用性，我就贴出如下：

```
SQL>select t1.sid, t1.value, t2.name
      from v$sesstat t1, v$statname t2
     where t2.name like '%user commits%'
           and t1.STATISTIC# = t2.STATISTIC#
           and value >= 10000
     order by value desc;
SID STATISTIC#          VALUE          NAME
-----
      991          4      2281122 user commits
--略去
SQL> select sid,username, t.PROGRAM,sql_id , t.PREV_SQL_ID from v$session  t where sid in =991;
SID  USERNAME          PROGRAM                                SQL_ID          PREV_SQL_ID
-----
  991 BOSSWG          timeTask@itsm_ht (TNS V1-V3)          avms342zm5k9u
SQL> select sql_text  from v$sql where sql_id='avms342zm5k9u';
SQL_TEXT
-----
update t set col1=:col1
```

为什么 `update t set col1=:col1` 这个语句会提交这么多次呢？联系相关开发人员后才明白，他居然把提交语句 `commit` 放在了 `loop` 循环的内部，导致每更新一条记录，就提交一次，循环 100 万次，就提交 100 万次，这是不应该犯的错误。

解决方法很简单，就是把 `commit` 提交放到了循环的外部，这下提交次数大幅度减少了。接下来系统运行就再也不会出现这个等待事件了，而且开发人员观察自己的这个更新语句，比之前快了几十倍！

其实快这么多是很正常的，这个例子和前一章节描述的因为索引导致插入变慢这一案例的

解决有异曲同工之妙。都是单次操作和批量操作的比较。每插入一条维护一次索引和让索引先失效，最后重建批量维护索引，在上次的案例中，居然也有千百倍的性能差异。

每插入一条提交一次，每提交一次系统就要让 redo buffer 写出一次，导致 lgwr 相当繁忙。关于批量操作，老师之前比喻的来回搬运砖头，一次拿一块砖和一次推一车砖谁更高效，我认为是最形象，最容易理解的。

看来少做事的案例真是多啊，同学们，你们都听得明白吗？”

“明白！”大家都回答得很大声。

“很好，我希望你们不只是明白，最好能把这个意识变成自己思想的一部分，那我今天这个分享交流的辛苦付出，就没有白费了。”

8.7 迁移越来越快了

“大家还记得在前一章节中描述过与迁移有关的案例吗，说了安徽某项目需要迁移数据库的相关案例，这个故事的描述从大家面对 2TB 数据量的数据库一筹莫展开始，到后来因为允许排除 800GB 的大表和发现有 800GB 的回收站数据笑逐颜开而结束。

其实故事还是有续集的。

一般来说，400GB 的数据在当前 16 个 CPU 以及良好存储的机器配置下，预计导出需要 1 小时，导入需要 2 小时，大致在 3 小时左右。一般来说 3 小时还是可以接受的，不过电信局方提出最好能控制在 2 小时左右完成导出和导入工作。因为越早完成迁移工作越安全，迁移成功后还有很多的工作需要去做，而早上 8 点上班前系统必须确保正常可用。

后来我倒是想出了一个主意提高了迁移的速度，说出来很简单，其实就是生活道理，和技术真的没关系，大家愿意不愿意听？”老师有些故弄玄虚了。

“愿意！”

“我先把数据库中的大表再次筛选了一次，总共捞出了连同索引大致有 200GB 左右的系列大表，准备单独导出这部分记录，我把这部分记录称之为 DMP2，而另一部分就是 DMP1，这部分其实就是之前准备导出的整个用户的大小，我用 EXCLUDE 语句排除了那些 200GB 左右的大表。这下我把原先的准备导出的 DMP 文件变成了 DMP1 和 DMP2。

我先从源数据库导出 DMP1，然后内网快速 FTP 到目标主机完成 DMP1 文件的 IMPDP 导入。接下来我再导出 DMP2，等 DMP2 导出结束后，再 FTP 到目标主机完成 DMP2 文件的 IMPDP 导入。

这和原先的一个 DMP 有什么差别呢？

差别就是，原先从源数据库导出，再导入到目标数据库，基本上不会存在两个机器同时工作的情况。而我切割了数据后，就会出现某个时段两台机器同时干活的情况。所以迁移速度必然有

改善。

结果还真是如我所愿，导出导入工作全部完成，仅仅花费了 1 小时 45 分钟，大大超出了原先 3 小时的预期值！

哎呀，糟糕了，老师在说少做事，怎么感觉这个案例在描述如何多做事啊，好好的把数据切割成 2 份，多辛苦啊，是不是这样？”老师笑着问。

台下大家都乐了。

“其实这也是少做事的另一种高级战术，合理利用资源。除了利用某段时间两台机器同时干活这个想法是在合理利用资源外，这次迁移我们根据当前机器有 16 个 CPU 的情况将迁移设置了 16 个并行度的这个手段，也是合理利用资源。合理利用外部资源提高工作效率，这显然也是少做事。

看来隐形菜刀比起前一章节的赤裸裸的直接菜刀，手法上确实丰富了许多，不过他们的本质都是一个：把握需求，尽量少做事！”

8.8 SQL 语句精简了

“同学们之前见过我在上一章中把 SQL 一刀剁了，整个模块都砍了的优化案例，听起来非常轻松愉快。

虽然这样的处理在我的案例中时常出现，但并不是所有场合都能这样痛快淋漓地处理问题，大部分时候还是要与一些不良 SQL 硬碰硬迎面交战。此外还有一点就是，我的课程中考虑到复杂语句是由简单语句组成的，并且为了突出知识本身的原理和意识的重要性，我给大家举例说明的 SQL 都是非常简单的。不过现实中，你面对的有性能问题的 SQL 语句有时写法是很恐怖的，部分开发人员的代码写得过于冗长、复杂、九转十八弯让人看得头晕眼花，无从下手。

为了让同学们知道江湖还是凶险的，我今天就完整地举一个复杂 SQL 语句的优化案例，这个 SQL 优化的核心还是在于少做事的意识，用隐形的菜刀将复杂的 SQL 精简为简单的写法，将原先的多次表扫描减少为一次表扫描。

大家做好准备，请看这个超长 SQL 语句（注意，为了缓解本恐怖片的紧张程度，笔者已经删减了其中部分情节，具体删除的镜头详见后续说明），具体如下：

```
select distinct ne_state.peer_id peer_name,
               to_char(ne_state.ne_state) peer_state,
               (case
                 when ne_state.ne_state = 0 then
                   to_char(0)
                 else
                   (select distinct to_char(nvl(ne_active.active, 0))
```

```

        from dcc_sys_log,
        (select peer_id,
            decode(action,
                'active',
                1,
                'de-active',
                0,
                0) active,
            max(log_time)
        from dcc_sys_log
        where action = 'active'
        or action = 'de-active'
        group by (peer_id, action)) ne_active
    where dcc_sys_log.peer_id = ne_active.peer_id(+)
    and dcc_sys_log.peer_id = ne_state.peer_id)
end) peer_active,
(case
    when ne_state.ne_state = 0 then
        to_char(0)
    else
        (to_char(nvl((select count(*)
            from dcc_ne_log
            where dcc_ne_log.result <> 1
            and peer_id = ne_state.peer_id
            and log_time between
                trunc(sysdate) and sysdate
            group by (peer_id)),
            0)))
end) err_cnt,
(case
    when ne_state.ne_state = 0 then
        to_char(0)
    else
        (to_char(nvl((select count(*)
            from dcc_ne_log in_dnl
            where in_dnl.direction = 'recv'
            and in_dnl.peer_id =
                ne_state.peer_id
            and log_time between
                trunc(sysdate) and sysdate),
            0)))
end) recv_cnt,
(case

```

```

when ne_state.ne_state = 0 then
  to_char(0)
else
  (to_char(nvl((select sum(length)
                from dcc_ne_log in_dnl
                where in_dnl.direction = 'recv'
                and in_dnl.peer_id =
                  ne_state.peer_id
                and log_time between
                  trunc(sysdate) and sysdate),
                0)))
end) recv_byte,
(case
  when ne_state.ne_state = 0 then
    to_char(0)
  else
    (to_char(nvl((select count(*)
                    from dcc_ne_log in_dnl
                    where in_dnl.direction = 'send'
                    and in_dnl.peer_id =
                      ne_state.peer_id
                    and log_time between
                      trunc(sysdate) and sysdate),
                    0)))
  end) send_cnt,
(case
  when ne_state.ne_state = 0 then
    to_char(0)
  else
    (to_char(nvl((select sum(length)
                    from dcc_ne_log in_dnl
                    where in_dnl.direction = 'send'
                    and in_dnl.peer_id =
                      ne_state.peer_id
                    and log_time between
                      trunc(sysdate) and sysdate),
                    0)))
  end) send_byte
from dcc_ne_log,
(select distinct dsl1.peer_id peer_id,
  nvl(ne_disconnect_info.ne_state, 1) ne_state
from dcc_sys_log dsl1,
(select distinct dnl.peer_id peer_id,

```

```
        decode(action,
                'disconnect',
                0,
                'connect',
                0,
                1) ne_state
    from dcc_sys_log dsl, dcc_ne_log dnl
    where dsl.peer_id = dnl.peer_id
    and ((dsl.action = 'disconnect' and
        dsl.cause = '关闭对端') or
        (dsl.action = 'connect' and
        dsl.cause = '连接主机失败'))
    and dsl.log_time =
        (select max(log_time)
         from dcc_sys_log
         where peer_id = dnl.peer_id
         and log_type = '对端交互')) ne_disconnect_info
    where dsl1.peer_id = ne_disconnect_info.peer_id(+) ne_state
    where ne_state.peer_id = dcc_ne_log.peer_id(+)
```

这条 SQL 语句原本是一条 crontab 定时任务，每 5 分钟执行一次，可惜了这语句跑了近 1 小时才跑出结果，导致每隔 10 分钟新启动一个脚本执行，但是原先的一直没有结束，最后把主机的 CPU 资源耗光了。

请同学们花两分钟时间看看贴出来的这个 SQL 语句，是不是很恐怖，我们姑且先不讨论执行计划，也暂不去了解这个语句最终返回多少条，虽然说这两点是 SQL 优化最先需要了解的东西。今天我们就这个 SQL 写法的本身来看看，是否存在问题。

请大家认真看两分钟，我开始计时。”老师抬起手看了看手表。

“老师，我觉得写法有点奇怪，这个语句好像要展现 err_cnt、recv_cnt、recv_byte、send_cnt 和 send_byte 这 5 个列，都是针对 dcc_ne_log 表的查询，写法都非常相似，差别在于有的地方是 SUM 有的是 COUNT，再有就是某一个条件略有差别。”一分钟后晶晶起身回答。

“晶晶同学观察得非常仔细，这个写法看似非常复杂，其实代码中有不少地方相似度极高。其实需求很好理解，开发人员得到错误条数 err_cnt、接受条数 recv_cnt、接受字节数 recv_byte、发送条数 send_cnt 和发送字节数 send_byte，为了得到这 5 个列而把这个 dcc_ne_log 表访问了 5 次，现在同学们看清楚这部分含义后，是不是觉得也不怎么晕了？所以善于观察也是很重要的。

这里我偷偷透露一下，其实真正的源代码中获取了的类似列有 10 个！比如还有最大条数，最小条数等等，我故意把这些在代码中略去了，这就是前面我说的删减的部分镜头。”

台上传来一片笑声。

“其实这时候我们应该考虑合并的写法，SQL 开发中有一个非常有用的语法，就是 CASE

WHEN，我们可以用这个语句来合并刚才这 5 个字段，经过适当的转换，构造这 5 个列（其实是 10 个列哦）的写法改造如下：

```
select peer_id
      ,COUNT(CASE WHEN RESULT <> 1 THEN 1 END) err_cnt
      ,COUNT(CASE WHEN direction = 'recv' THEN 1 END) recv_cnt
      ,SUM(CASE WHEN direction = 'recv' THEN length END) recv_byte
      ,COUNT(CASE WHEN direction = 'send' THEN 1 END) send_cnt
      ,SUM(CASE WHEN direction = 'send' THEN length END) send_byte
      ----其实还有 5 个被略去了，类似最大，最小记录等
from dcc_ne_log
where log_time >=trunc(sysdate)
GROUP BY peer_id
```

这部分大家看得明白吗？”老师问。

台下纷纷点头。

“最终整体 SQL 语句改写完毕后（大家重点关注 COUNT 或者 SUM 与 CASE WHEN 结合部分的写法，其他部分的调整暂且不要去关注），代码量大大减少的同时性能极大地提升了，表扫描次数大大缩减了，执行完成时间从原先的 3000 秒缩短为 5 秒。完整改造后的最终 SQL 代码优雅精致，具体如下：

```
with ne_state as
(SELECT a.peer_id,
CASE WHEN dnl.peer_id IS NOT NULL AND str IN ('disconnect 关闭对端','connect 连接主机失败') THEN '0' ELSE '1'
END ne_state
FROM (SELECT peer_id,MIN(action||cause) KEEP(DENSE_RANK LAST ORDER BY log_time) str
      FROM dcc_sys_log dsl
      WHERE log_type = '对端交互'
      GROUP BY peer_id
) a,(SELECT DISTINCT peer_id FROM dcc_ne_log) dnl
WHERE a.peer_id = dnl.peer_id(+)),
dcc_ne_log_time as (select peer_id
                    ,COUNT(CASE WHEN RESULT <> 1 THEN 1 END) err_cnt
                    ,COUNT(CASE WHEN direction = 'recv' THEN 1 END) recv_cnt
                    ,SUM(CASE WHEN direction = 'recv' THEN length END) recv_byte
                    ,COUNT(CASE WHEN direction = 'send' THEN 1 END) send_cnt
                    ,SUM(CASE WHEN direction = 'send' THEN length END) send_byte
                    from dcc_ne_log
                    where log_time >=trunc(sysdate) ---- between trunc(sysdate) and sysdate
                    GROUP BY peer_id)
select distinct ne_state.peer_id peer_name,
               to_char(ne_state.ne_state) peer_state,
```

收获，不止 Oracle

```
(case
  when ne_state.ne_state = 0 then
    to_char(0)
  else
    NVL((select '1' from dcc_sys_log where peer_id = ne_state.peer_id and action =
'active' and rownum=1),'0')
end) peer_active,
decode(ne_state.ne_state,0,'0',nvl(dnlt.ERR_CNT,0)) ERR_CNT, ---注意 NVL 改造
decode(ne_state.ne_state,0,'0',nvl(dnlt.recv_cnt,0)) recv_cnt,
decode(ne_state.ne_state,0,'0',nvl(dnlt.recv_byte,0)) recv_byte,
decode(ne_state.ne_state,0,'0',nvl(dnlt.send_cnt,0)) send_cnt,
decode(ne_state.ne_state,0,'0',nvl(dnlt.send_byte,0)) send_byte
from ne_state ,dcc_ne_log_time dnlt
where      ne_state.peer_id=dnlt.peer_id(+)
```

第 9 章



提问，也是智慧的体现

提问的智慧所涉及的内容如图 9-1 所示。

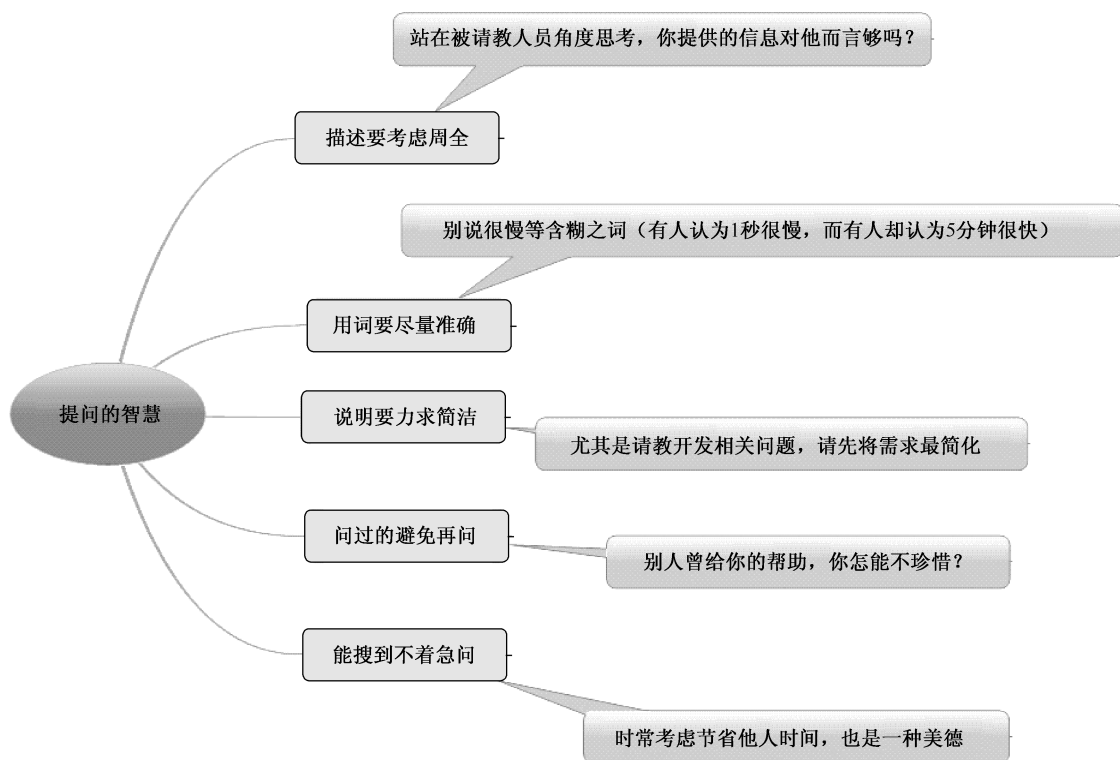


图 9-1 提问的智慧

“同学们，今天这节课我们不讨论技术，但是却是非常重要的一课，甚至比前面的系列 Oracle 知识更为重要。

那我们讨论什么呢？就是关于提问的智慧。

每个人的能力精力有限，且术业有专攻，谁都不可能精通所有相关知识，而问题往往有可能

涉及多领域，因此遇到自己难以独立处理应对的场景时有发生！此时少钻牛角尖转向及时求救是非常必要的。要如何提问呢？

下面通过几个有趣而又真实小案例，来分析探讨我们该如何正确提问。

9.1 描述要考虑周全

某天早上 9 点，我收到了省外工程点小 A 的一封求助邮件，原文如下。

梁老师：

黑龙江 XX 系统的数据库有故障，请您帮忙处理一下。谢谢！

此致

敬礼！

小 A

同学们，你们收到这样的一封邮件，作何感想呢？”

“老师，这个邮件写得也太简单了吧。”小莲差一点笑出声来。

“说得很好，就是描述得太简单了。小 A 犯的最大的错误就是只站在自己的角度看事物，习惯性地认为自己懂的别人都明白了，没有把事情描述清楚。不过大家别笑，你们当中也有不少人求助问题时描述不周全的。

小 A 要别人帮他解决问题，首先他要清楚地告诉人家遇到了什么问题。邮件中的‘故障’这两个字眼让人摸不着头脑。不知道是数据库无法访问，还是数据库无法启动，还是数据库运行缓慢。如果他能清楚描述，别人就可以做到心中有数，根据经验给他相应的建议。比如他告诉别人数据库无法访问，别人会让他提供监听的相关日志及网络的相关信息，从而帮他解决问题。

好吧，即便他不说清楚是什么故障，那总要告诉别人如何访问他的环境吧，提供一下 IP 地址和用户名密码等，这样别人或许可以登录黑龙江的系统，从而帮他定位解决他所描述不清的问题。

无奈我想打电话直接找小 A 了解一下情况，拿起手机却发现邮件里没有提供电话联系方式，我没有花费更多的精力去多方了解小 A 的手机联系方式，因为通过邮件我不知道黑龙江的故障是生产的还是非生产的，影响面有多大。最后就回一封邮件，让他说清楚是什么故障，让他告知我如何才能访问他们的环境，并附上我的联系方式以便他直接电话和我交流。

直到上午 11 点我才接到小 A 的电话，知道了这是来自生产环境的严重故障。数据库主机 CPU 资源被耗尽导致生产系统运行缓慢，引起大量投诉。通过小 A 提供的登录方式，我连到了黑龙江环境，很快定位到问题所在，原来是某些执行频繁且运行缓慢的 SQL 引发的相关故障，这些耗尽 CPU 资源的 SQL 来自昨晚的补丁，回退这些补丁后，问题解决。

问题虽然解决了，可是有一个糟糕的事实让人难以接受。就是这次故障影响生产系统时间持

续了两个多小时了，影响相当大！

我至今不知道为什么小 A 要等两小时后才给我打电话，不过他如果当时能在邮件中说清楚是性能问题，并且提供了登录方式，我想我应该可以确保在 20 分钟内解决问题。如果他能再描述一下系统是由于昨天晚上打补丁后忽然不正常了，我就会让他们考虑立即回退补丁，或许在 10 分钟内就消除了故障。

还有，如果他能在邮件中说明是生产系统的故障，影响面很大，那我就会放下手中的活，千方百计找到小 A，也就不会有这难堪的一幕发生了。”

9.2 用词要尽量准确

“又收到一封求救的邮件，来自小 B 的，原文大致如下。

梁老师：

宁夏某生产环境有一条 SQL 语句自 5 月 10 日上线以来执行一直非常慢，影响了系统的正常使用，请您帮忙看看，能否优化。

SQL 语句和访问宁夏生产环境登录方式及联系人信息详见附件，谢谢！

此致

敬礼！

小 B

看来小 B 要比小 A 描述问题清楚多了，我登录到宁夏环境执行了该 SQL 后，惊奇地发现只需要 1 秒左右就可以运行结束。奇怪，这还慢？

后来和小 B 确认后才知，原来这个语句一天需要执行上千万次，如果不能在 0.01 秒内完成，将无法完成任务，原来 1 秒也是非常慢的。回想起前段时间的一次优化经历，我将某 SQL 从原先的 40 分钟优化到在 5 分钟内执行完毕，求助人员开心地忍不住拍起手来，说 5 分钟能完成，真是太快了！

我纳闷了，同一个公司的同事，有的说 1 秒钟执行完毕太慢了，而又的却因为 5 分钟可以执行完毕而开心地拍手。所以如果有人需要我优化 SQL，我不希望听到‘非常快’和‘非常慢’这样的模糊字眼，直接告诉我你当前执行时间多长，需要优化成多长时间完成，就好了。

这里我要说明是，问问题，尤其是在 IT 领域，用词要尽量准确。还有类似的例子是，有人告诉我数据库崩了，该怎么处理。我一脸茫然，什么叫崩了？经过多次确认才明白他说的崩其实是描述系统压力大，CPU 的空闲率为 0。如果非要让我对数据库崩了做一个解释，我理解的崩可能是数据库停止运行，无法启动，和他的描述差异显著。为什么不说清楚点，这很难吗？多余的交互让我们浪费了多少宝贵的时间啊。”

9.3 说明要力求简洁

“前面一直说描述要周全，但是周全并不等于啰嗦，周全和简洁一点都不矛盾。把问题描述简洁，让人快速明白，这是非常重要的。

有一次某生产系统揪出一条运行效率十分低下的 SQL 语句，由于该语句极其复杂冗长，所以需要从业务层面来分析，希望改造代码来优化。

写该 SQL 的开发人员小 C 不知该如何改写，由他出面来描述需求到底是什么。结果该项目组小组成员整整交流了 2 个小时，还没弄清楚小 C 的业务逻辑到底是什么意思。

后来小 C 直接来找我了，让我帮忙改写一下 SQL，听了他的描述 10 分钟后我就不耐烦了，说了一堆的业务背景和一堆的表名和列名，听得我一头雾水。

最后我打断了他的描述，亮出了自己的杀手锏，让他写一个最简单的表名，然后写进你需要的列名，然后构造一些数据，直接告诉我，你想显示出什么样的结果。

在我的要求和监督下，他照做了。5 分钟后，我立即明白他在说什么了，再有 5 分钟，我帮他改写好 SQL 了，然后他根据我的语句将表名和列名对应调整一下，SQL 改造好了。

```
DROP TABLE TEST;
CREATE TABLE TEST ( ID1 NUMBER,ID2 NUMBER,VALUE1 VARCHAR2(20),VALUE2 VARCHAR2(20));
INSERT INTO TEST VALUES (1,2,'A','B');
INSERT INTO TEST VALUES (1,2,'C','D');
INSERT INTO TEST VALUES (1,2,'E','F');
INSERT INTO TEST VALUES (1,2,'G','H');
INSERT INTO TEST VALUES (3,8,'I','J');
INSERT INTO TEST VALUES (3,8,'K','L');
INSERT INTO TEST VALUES (3,8,'M','N');
INSERT INTO TEST VALUES (8,9,'O','P');
INSERT INTO TEST VALUES (8,9,'Q','R');
INSERT INTO TEST VALUES (11,12,'S','T');
COMMIT;
```

```
SQL> SELECT * FROM TEST;
```

ID1	ID2	VALUE1	VALUE2
1	2	A	B
1	2	C	D
1	2	E	F
1	2	G	H
3	8	I	J
3	8	K	L
3	8	M	N

8	9	O			P		
8	9	Q			R		
11	12	S			T		

10 rows selected

要求显示出如下结果
要求为(行列转换，超过 3 个的只取三个，不足 3 个的用空格来补列)

ID1	ID2	VALUE1	VALUE2	VALUE3	VALUE4	VALUE5	VALUE6
1	2	A	B	C	D	E	F
3	8	I	J	K	L	M	N
8	9	O	P	Q	R	NULL	NULL
11	12	S	T	NULL	NULL	NULL	NULL

到这里，应该所有的人都能理解含义了，接下来我的如下语句实现了这个需求：

```
SELECT ID1,ID2
      ,MAX(DECODE(RN,1,VALUE1))
      ,MAX(DECODE(RN,1,VALUE2))
      ,MAX(DECODE(RN,2,VALUE1))
      ,MAX(DECODE(RN,2,VALUE2))
      ,MAX(DECODE(RN,3,VALUE1))
      ,MAX(DECODE(RN,3,VALUE2))
  FROM (SELECT TEST.*, ROW_NUMBER() OVER(PARTITION BY ID1,ID2 ORDER BY VALUE1,VALUE2) RN FROM
TEST) T
WHERE RN<=3
GROUP BY ID1,ID2;
```

有兴趣的同学们完全可以自行试验一下输出的结果是否正确，是否符合要求。后来我改写的语句在 5 秒之内可以完成，而原先小 C 的语句执行了近 2 个小时才运行完毕。为什么差别如此之大，原因在于小 C 的 SQL 逻辑都错了！

看来需求最小化不只是方便自己请教他人，还可以让自己验证代码的准确性。我们所在的项目组，如果有人问我 SQL 如何实现，我会让他们以最简洁的方式来提问，提供建表语句，构造数据，然后告诉我，他想展现什么样的结果。

能否简洁地描述是非常重要的，这是在简化大家的工作，让大家少在这个无谓的交互上做事！”

9.4 问过的避免再问

“‘梁老师，请帮忙看看，这个 SQL 该如何写，我想要……该如何实现呢？’小 D 通过 QQ 再次问我问题了。

‘这个问题你以前问过了，可以这样这样解决……’答复他的同时，我无可奈何地摇摇头。

‘有吗？我怎么一点印象都没有啊。’小 D 似乎有些不相信。

‘你上次是发邮件问我的，你稍等一下，我转给你。’

‘哇，不是吧，真的问过一样的问题了，您当时还回答这么详细。不好意思，我再看看，多谢多谢！’

好了，这是不是一段有趣的对话啊？”梁老师说到这里，停下来问大家。

台下同学们都笑了。

“这里我强调一点，问过的问题尽量不要再问。如果你重复问同样的问题，可以很明确地说明你当时并没有深入思考，或者说并没有真正理解，这种不动脑筋的学习态度注定了你成不了技术能手，同时你也在浪费别人的宝贵时间。

小 D 是一个非常勤奋好学之人，经常请教大家技术问题，可是他的技术水平在项目组却不受大家的认可，工作完成的也不怎么好，这显然和缺乏思考盲目请教他人的习惯有关。

我们再举一个例子。

某天小 D 遇到一个技术问题要请教我，由于我当时处理紧急故障无暇帮他，就让他自己上网搜索或者在论坛上提问一下，以获取别人帮助。

很快小 D 就喜上眉梢地告诉我，有人回复了他在 ITPUB 上发的求助帖，他的问题解决了。不过后来我闲下来时看了他的求助和别人回复的内容后，忍不住笑出声来了。

原来这个热心的回复者是这样回复的：这个问题你之前已经问过一次了，我也回复过你了，请看这个链接。随后附了一个之前小 D 发的老帖子的超链接。

后来我找小 D 好好聊了一次，我对他说，只要你能做到问过的问题不再问，你的技术水平就能迅速提高一大截。并且我还和他打了个赌，以三个月为期限。

令人高兴的是，我打赌赢了，小 D 工作中的表现很快有了改观，受到项目组一致表扬后他很高兴地输给了我一顿饭。不过最后买单的是我，因为看到项目组成员的进步是最快乐的事情，谁请客是次要的，这是一次庆功宴，不是一场打赌。”

9.5 能搜能试不急问

“小 E 是我见过的最会请教他人的同事了，几乎是一有问题就请教他人。可是他请教我的时候却时常让他不满，为啥会这样呢，请看看我们之间的对话。

‘梁老师，建分区表的 HASH 分区如何建，语法怎么写？’

‘我忘记了，你搜搜看吧。’

‘梁老师，建全局临时表的语法是哪个关键字啊，我记不起来了。’

‘我也记不起来了，你搜搜看吧。’

‘梁老师，我想对表建一个位图索引，该怎么写？’

‘你自己搜一下吧。’

‘梁老师，分区表可以建范围和 HASH 的组合吗？’

‘我忘记了，你试试看。’

‘梁老师，我听说对表建索引会把表锁住，是真的吗？’

‘你有时间可以试试啊。’

这就是我们之间的对话。后来我听说小 E 私下经常对人说，梁老师的技术水平好像也不怎么样，我问他的问题，他不是让我搜，就是让我试，都不能直接回答我。”老师说到这里，露出了一脸的无辜。

台下同学都哈哈笑了。

“其实小 E 同学的问题，有些我是真回答不上来，比如 HASH 分区的语法和全局临时表的语法。可事实上记不住这些又有什么关系呢，不是很方便可以搜到吗？关键是要掌握这些知识的原理，思考过这些知识适用的场景，并和业务结合起来，这才是最重要的。

记忆力再好的技术人员也不会记住全部的语法和命令，可以允许我们查询、翻阅。能力的高低不是体现在语法的记忆上，而是在知识的理解和应用上。我们的精力是要花在灵活应用知识，解决实际问题 and 不断创新上面，而不是用在记忆语法上。

此外动手能力是非常重要的能力，在动手过程中，你要构造环境，就必须将各种知识组合在一起，这是难得的知识组合学习的过程，在探索过程中，对知识的印象也会加深。小 E 问的组合分区和索引锁表的问题，完全可以通过自己动手试验来证明。然而小 E 却并不想这样做，只希望从别人口中获取答案。这不仅仅是缺少了锻炼，万一别人的回答是错误的，该怎么办呢？”

第 10 章



买鱼，居然买出方法论

10.1 小余买鱼系列故事

“同学们，带意识的 Oracle 学习我们暂时告一段落了，希望大家将来的学习中能时刻记得学什么、什么原理以及如何应用的这三句箴言。这样能让人在学习过程中少做很多无谓的事，从而高效地完成学习，不只是 Oracle 学习，各个领域的学习也都是如此。

那接下来的课程，进入了思想与案例分享的阶段，同学们将会更加轻松和随意。老师的系列分享将从一个有趣的故事《小余买鱼》开始，这依然是老余一家的故事的延续。这个故事和优化有关，相信会给大家带来很大的启发。”

台下又是一片欢呼雀跃。

“那我开始讲述了。

小余是一个聪明的孩子，善于动脑，为父亲母亲在生意经营上出了诸多好主意。由于勤劳且善于思考，现在已经是成长为一名侦探了。不过人无完人，他也有处事考虑不周之时，有关他的故事，从他曾经的买鱼经历开始说起。”

10.1.1 诊断与改进

“一天下午 4 点多，小余妈妈想做水煮活鱼给家人吃，让小余去买一只草鱼回来。小余骑自行车到 20 里外的沃尔玛超市买到鱼然后返回。一到家，妈妈就开始责怪小余买鱼花的时间太长了，因为都已经是下午 6 点半了，晚上 7 点一家人都安排好了外出的活动了，这下水煮活鱼来不及做了……

故事说完了，好听吗？”

“老师，这故事没什么情节啊！”小莲一句话引发大家的笑声一片。

“虽然看上去这个故事既没内容又不精彩，但是在我看来，Oracle 的优化思想、方法论却在这个小故事中展现得淋漓尽致。你们看出来了吗？”老师问大家。

从同学们的表情来看，大家都有些茫然。

“不明白？让我们慢慢展开分析。

很明显妈妈对小余买鱼的工作效率不满意（导致她做鱼的时间都没了），问题出在哪呢？小余是个聪明的孩子，善于思考，这次挨批后，他想好好反省总结一下，避免将来再犯同样错误而再次挨批。

他想**诊断**出问题出在哪个环节，为此他得先思考具体都有哪些环节，这就是：

将过程细化

小余回想买鱼经历后，将整个买鱼过程做了细化，具体哪些环节马上出来了：

- ① 骑车来回往返
- ② 银行排队取钱
- ③ 在超市里找卖草鱼的地方
- ④ 其他时间开销（如取自行车、停车、接电话等）

接下来他很自然地将思绪飘落到如下之处。

找细化项主要矛盾（开销最大的部分）

依次分析各细化项的开销：

① 骑车往返花费	90 分钟
② 钱不够，去银行取钱排队花费	20 分钟
③ 在超市里到处找卖草鱼的地方，这些动作花费	20 分钟
④ 其他时间花费（如取自行车、停车、接电话等等）	5 分钟

小余猛一拍脑袋，重大突破来了，主要时间开销在买鱼往返途中，主要矛盾在这！如果能大幅度缩短往返时间，必然能让妈妈少等许多时间。

小余接着开始考虑怎么**优化改进**，从而解决问题。他脑海瞬间闪过一点：妈妈让我做什么事？这就是：

准确理解需求。

妈妈的需求是什么呢？一句话：需要买来草鱼做晚餐。妈妈并没有要求小余一定要在哪买鱼，只要能买到，就满足妈妈的需求了。小余意识到自己犯了一个很低级的错误，他缺乏下面一个意识：

尽量少做事甚至不做事完成需求。

在妈妈没有要求在哪买鱼的情况下，小余去了 20 里外的超市买鱼，小余忽然觉得自己是不是有点傻。20 里外的地方是他学校，平时上学时他留意到附近有一家大超市，同时他无法确定自己家附近是否有可以买到鱼的超市，这正是他要去如此远的地方买鱼的原因。

如果家附近能买到鱼却选择去 20 里以外的超市买，岂不是多做事完成同样的需求了？

小余意识到明确自己家附近能否买到鱼，将是处理问题的关键！经过自己的调查研究，他发现自己家附近真有可以买到鱼的农贸市场，虽然农贸市场很小，但是不影响自己完成买鱼的任务，专业的说法是，并不影响需求的实现。

他开始特别懊悔自己预先不问问妈妈或者其他知情人，否则这 90 分钟的往返路程是完全可以缩短为几分钟的哦。

一周后妈妈又要做水煮活鱼了，小余就在楼下附近农贸市场买了鱼，这次他没再让妈妈等太久，妈妈对小余买鱼的效率非常满意。

到此问题圆满解决了，小余冷静思考生活中的小事，总结出了诊断问题和解决问题的方法论如图 10-1 所示：

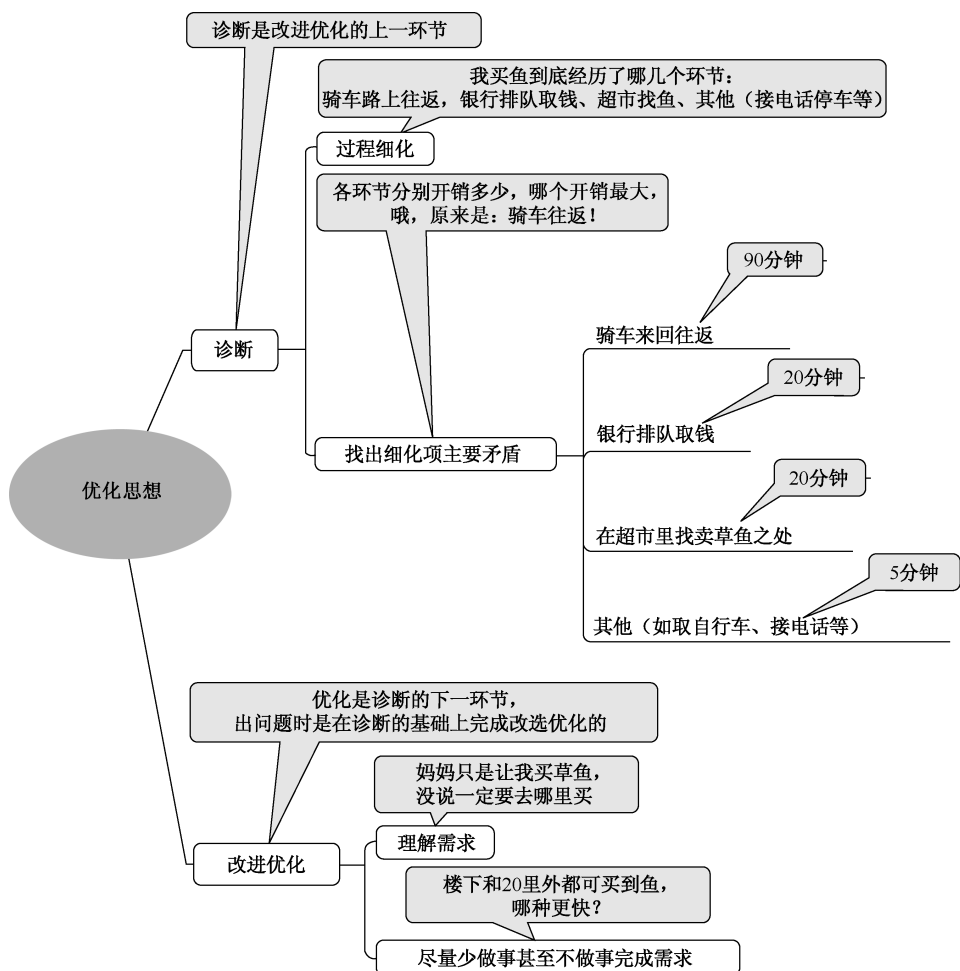


图 10-1 优化思想

通过此事，小余成长了许多！生活和技术是相通的，生活如此，Oracle 优化也是如此，同学们，你们现在能感受到 Oracle 的优化方法论尽在其中吗？”

台下同学听得津津有味，对老师的观点纷纷表示赞同。

10.1.2 需求与设计

“现在觉得这个故事好听吗，还觉得没情节吗？”老师笑着问。

“好听！”同学们半开玩笑地回答得很大声，小莲有些不好意思地笑了。

“那我接着再讲《小余买鱼 2》了，大家听吗？”老师有些故弄玄虚。

“听！”

“一个月后，小余妈妈又准备做水煮活鱼了，妈妈还让小余去买一只草鱼回来。不过这次情况发生了变化了，家附近的农贸市场因故关闭了，由于住的比较偏僻，还真的只能去 20 里外的沃尔玛超市买鱼了。

如果是以前，小余必然是兴冲冲地一头冲出门，帮妈妈买鱼去。不过有了第一次买鱼的经历后，他学会了思考，变得更成熟了……同学们，此处略去 3000 字。

‘妈妈，我回来了！’妈妈看到小余提着鱼，连连称赞，非常满意。

故事说完了，好听吗？”

“老师，这是什么故事啊，更没内容啊！”晶晶忍不住起身说。

“我把这个故事说成悬疑小说了，也难怪大家急了。故事说完了还有很多疑问，咋办？要不，我还是代表一下广大听众，亲自采访一下小余本人，以下是精彩的采访片段。

‘你好，小余，很多读者都急切地想知道你第二次买鱼做了哪些改进，可以让你妈妈那么满意，谢谢！’

‘很乐意接受你的采访，谢谢！对了，我第一次买鱼失败经历后做了一些总结了，你知道吗？’

‘小余，你考我啊，我还记得。你是把握了**诊断**和**优化改进**两个方向。通过问题细化和抓重点这两个方式，诊断出需要解决问题的环节，然后结合理解需求和少做事原则，发现了舍近求远买鱼这个主要错误，水到渠成得出就在家附近买鱼这个改进方法。是这样吧？’

‘非常不错！’小余忍不住赞了一句，‘不过诊断是用在出问题的场合，我这次买鱼从头到尾都很顺利，也就不需要有诊断了。’

‘小余，可是这次情况有变化了，你家附近农贸市场关闭了，你只能去 20 里外的老地方买鱼了，你还能有啥好方法啊？’

‘优化方法论是诊断和优化改进两部分组成的，如果优化改进考虑非常周全让一切都进行得很顺利，诊断就会因为无用武之地而变成是多余的了，但是优化改进的思想却断然不存在多余的场合！经过上次失败后，我不断思索改进，现在我的想法比第一次总结的思想更全面合理了。’

‘小余，你有比第一次总结更全面合理的想法了？太好了，说来听听。’

‘主要改动在于，我认为改进优应该从两方面着手，一是理解需求，二是设计。具体看我画的草图 10-2。

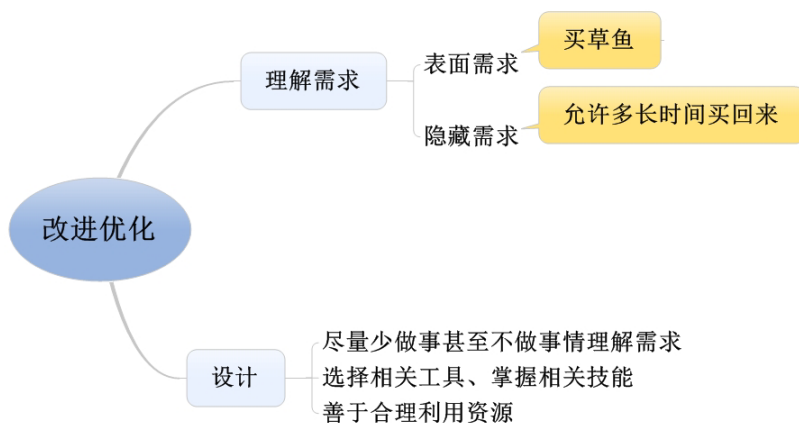


图 10-2 优化改进

其实我第一次买鱼失败还有另外一个重要原因，那就是无法准确把握隐藏的需求。这个隐含需求就是：妈妈没要求多长时间买回鱼，这个时长留给我自己去判断。不过很明显到下午 6 点多肯定是不合理的，因为耽误了吃晚饭。当然除了自己判断我们也可以去问妈妈多长时间合适，因此我把理解需求细分出表面需求和隐藏需求。

这次买鱼我问了妈妈具体需要多长时间，答复是下午 4 点出发，争取在 5 点前回来。哇，去同样的地方做同样的事情，要从 135 分钟缩减到 60 分钟？

任务该不该接下来呢？好好设计规划是必要的。上次仅往返途中就用了 90 分钟，必须先解决这个问题。我首先想到是否有近路去同样的目的地，并当即求助于对门邻居的士司机老王，他告诉我从刚开通的跨江大桥到沃尔玛超市，可以缩短一半以上的路程，并且答应陪我一起去。恰好我爸爸出差没用车，我这次让老王帮忙开爸爸的车子同去。原先的自行车走远路改为开车走近路的规划后，我们预计路上往返 30 分钟时间就够了（事实果然如此）。

根据之前的经验，银行排队花费了 20 分钟，这个完全没必要，这次带足钱去了。

而超市买鱼时找鱼花费了 20 分钟，也不应该，我之前居然没注意到其实超市有分类标志，如生鲜区、副食品区、用品区、水果区、服装区等。买鱼可以直接进生鲜区查找，接下来在生鲜区找到水产品区，在水产品区中再找到鱼类区，草鱼就买到了，大大缩小了找鱼的范围。

规划好后，觉得 60 分钟完成任务没问题，就愉快地答应去买鱼了，最后圆满完成任务！’

‘太精彩了，受益匪浅，多谢你，小余。’

采访结束了，同学们是否觉得很精彩，受益匪浅？”老师一会儿模仿自己的声音，一会儿模

仿小余的声音回答，过足了一把戏瘾后问大家。

同学们感触颇深。

“现在我们用下面这个完善改进优化的图表来结束《小余买鱼 2》的故事吧，如图 10-3 所示：

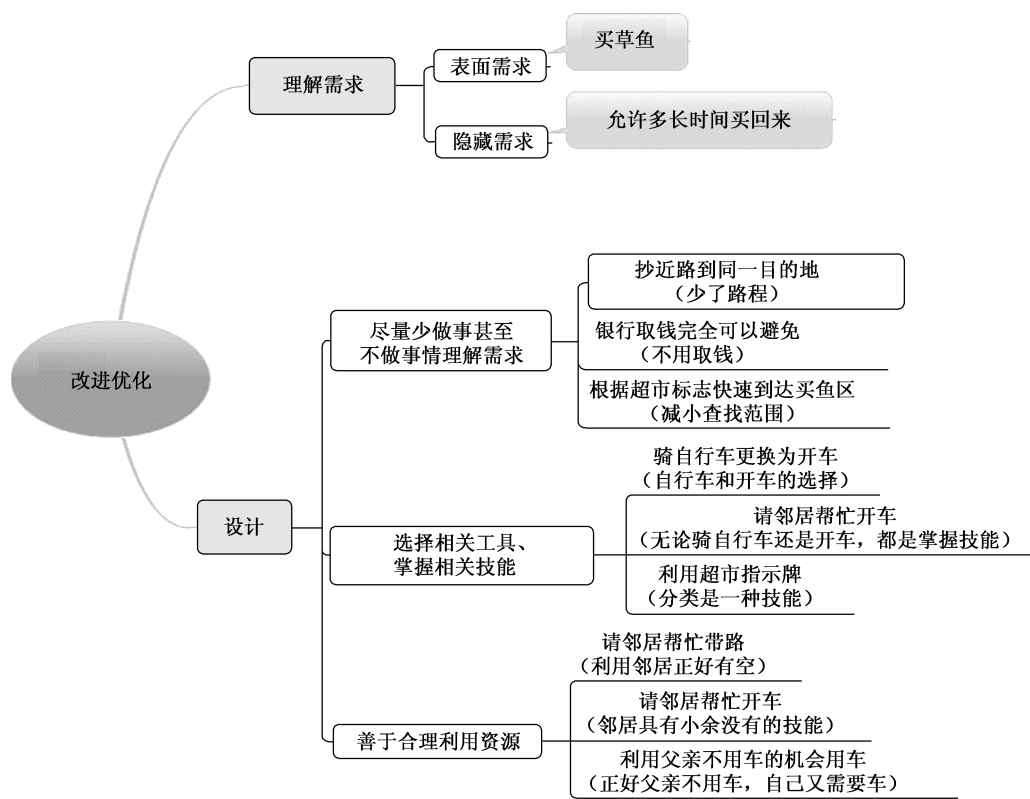


图 10-3 优化改进细节

这个图老师整得很辛苦，希望你们能仔细揣摩思考，不要辜负了老师的良苦用心。”

10.1.3 资源的利用

“由于小余善于总结，经历过两次买鱼的经历后，成长了不少，而且第二次完成得相当漂亮！接下来他妈妈又让他去买鱼了，大家觉得这次小余买鱼应该不会出现任何不合理之处了吧？”

台下纷纷点头。

“不过实际情况却很出乎大家的意料，这次小余居然又犯错了，让我们接下来听老师的《小余买鱼 3》。

又过了几天，妈妈再次让小余去买鱼，这次楼下附近的农贸市场开放了。小余没能摆脱上次买鱼延续的思维惯式，选择让表哥帮忙一起开车去买鱼，结果到地下车库开车、到了农贸市场找车位停车，花费了大量时间，导致比走路去还花费更多时间。这就是要注意什么场景选择什么样的处理方式（从技术角度来看就是什么时候选择什么技术）。也就是图 10-3 中设计的第 2 点再次强调，这是非常重要的。

事实上事情还更糟，小余买鱼期间爸爸老余正准备去参加紧急会议，结果车被开走了，最后导致会议迟到了。爸爸迟到这个事和图 10-3 设计中的第 3 点相关：善于合理利用资源。前一次，一来爸爸老余去外地了，二来买鱼的路途遥远，当然要合理利用资源。而情况变化后，就要及时考虑清楚了，车开走了，别人需要怎么办，你事先沟通过了吗，你想过了吗？”老师说到这里，停了下来，看到大家听得很仔细，很是欣慰。

“《小余买鱼 3》没人起来批评老师啊，我还以为又要被人批评了。”梁老师调侃地说，“此外，《小余买鱼 3》没有新的输出图，故事只是《小余买鱼 2》的一个补充和说明，请读者们回过头好好看看《小余买鱼 2》中的图表吧。”

10.1.4 真正的需求

“小余买鱼说完三集了，现在要准备说最精彩的一集了，大家要认真听，别走神！”

老师的故弄玄虚让大家对《小余买鱼 4》有了不少期待，他还能买成啥样呢？

“又过了一个月，妈妈又准备让小余买草鱼来招待刚上门做客的大舅了，不过因为离晚饭时间很近了，妈妈希望能在 20 分钟内买好鱼，而此时家附近的农贸市场依然没有开放，小余判断，无论如何都不可能完成这个任务了，不过小余还是开动了脑筋。最终居然让妈妈满意地点头了。你们谁能猜到小余做了什么事吗？

我估计谁也猜不到这次小余怎么让妈妈满意了。让我来直接公布答案吧。答案就是：最终小余让妈妈别买鱼了，用冰箱里的牛肉做水煮肉片。

小余买鱼系列故事全文完，谢谢大家！”

“老师，这是投机取巧啊！”敬昱第一个喊出来。

“错，这绝对不是投机取巧！”

其实这次小余很不简单，居然让妈妈改变了需求，从买鱼改为了直接用冰箱里的肉。其实小余是真正成长起来了。他并没有帮妈妈改变需求，而是帮妈妈一起发掘了真正的需求！

妈妈的真正需求并不总是买鱼，而是要展现手艺，和家人共享美味。从这个角度来看，水煮活鱼和水煮牛肉是一样的，大舅对妈妈擅长的水煮牛肉和水煮活鱼是同样感兴趣的。

鱼很难在规定时间内买回来，而牛肉就在冰箱。真正的需求已经从‘我要买到鱼做水煮鱼’

的表象转化为‘我要做美味和家人分享’。小余再次让妈妈满意了。”梁老师说到这里，还有些激动。

“这个故事其实很重要，在后面的优化中将会有不少落地的案例来说明。这里对设计中的理解需求做了完善，出现了一个‘理解真正需求’的概念，具体见图 10-4 所示。

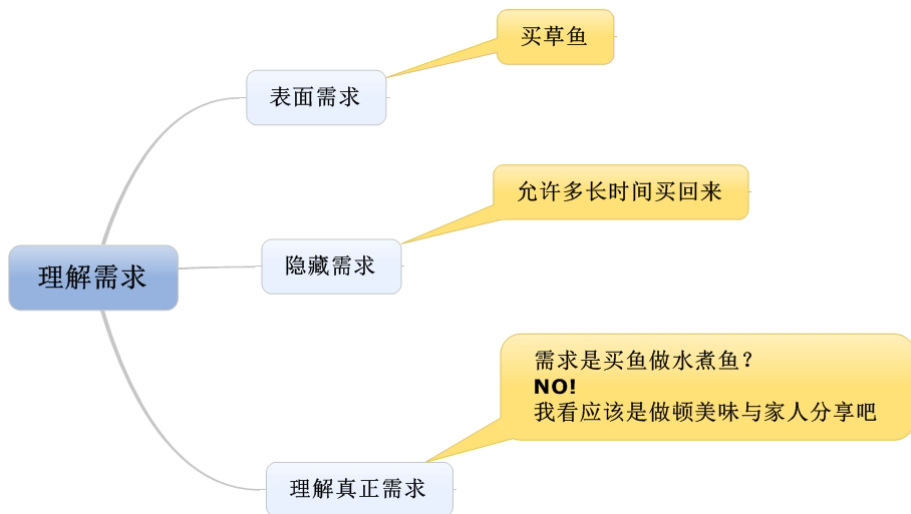


图 10-4 理解需求

至此，终于把小余买鱼这个重要故事说完了，大家好好体会 10 分钟，我们休息一下再继续。”

10.2 买鱼买出了方法论

10.2.1 一套流程

“至此可以用一套完整的流程图来暂时结束《小余买鱼》系列故事，此套流程概括了本文全部精华思想。浓缩了笔者多年的调优心得，该图（如图 10-5 所示）内容将在后续的描述中被不断印证！

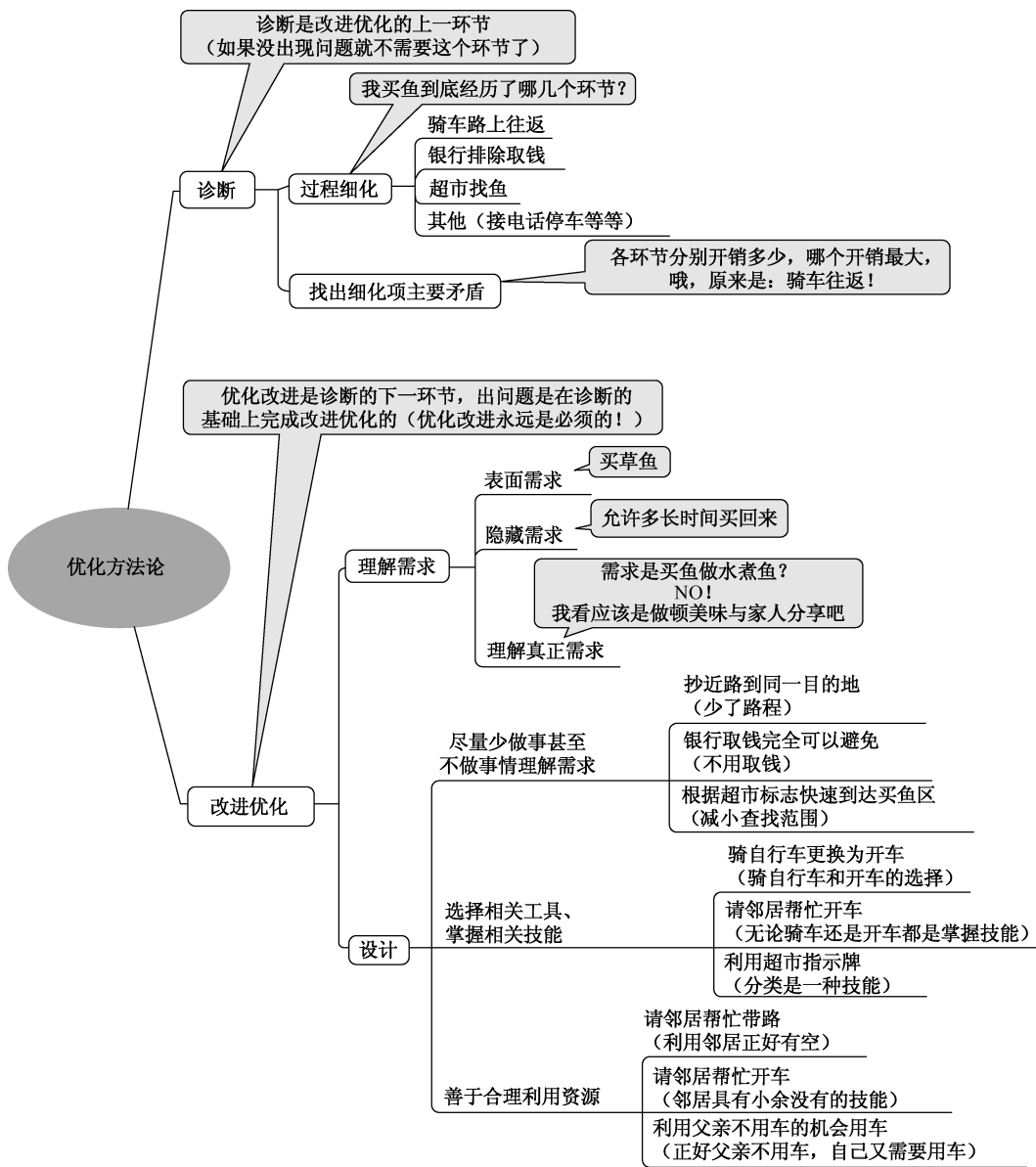


图 10-5 优化方法论

请大家仔细看大屏幕上这个图，生活和 Oracle 优化难道不是一样一样的吗？”

10.2.2 两大法宝

“上述总结的一套流程从小余买鱼系列故事演绎推理出来，从诊断和改进优化两个分支徐徐拉开优化方法论的序幕，形成了本文最宝贵精华部分。此外任何事物都可以多角度看待剖析。我们还可以将买鱼的事情分成意识（非技术能力）和技能（技术能力）两部分。在我们看来，这是优化的两大法宝（如图 10-6 所示）。

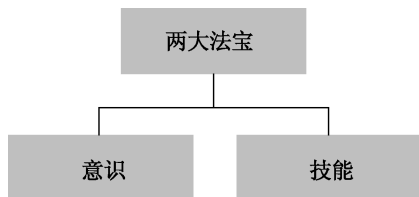


图 10-6 两大法宝

此话怎讲？让我细细道来。

小余思考自己买鱼具体经历了哪些环节、最长时间耗在哪个环节、楼下是否有鱼、如果一定要去某处买鱼是否有近路、自己去银行排队取钱能否避免……这些都属于意识类的，和具体的技术能力无关。小余买鱼过程中曾经骑自行车去买，也曾经开车去买鱼，无论是开车还是骑自行车，都不是一生下来就会的，必须经历过训练学习才能掌握，这就是技能类（如图 10-7 所示）。

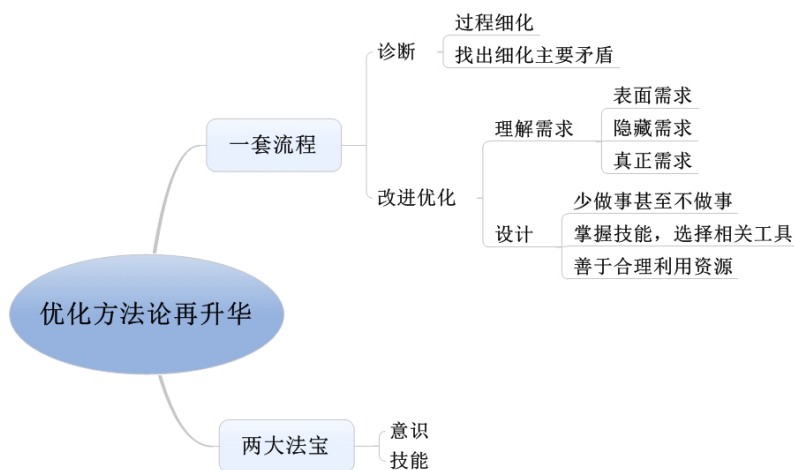


图 10-7 优化方法论升华

生活中的优化就是意识和技能的结合，两者都非常重要。首先说技能，掌握技能的重要性毋庸置疑，就好比要到很远的沃尔玛超市买鱼吧，你既不掌握开车的技能也没学会骑车的本领，而想靠走路到达目的地，那估计到达时店铺也打烊了吧。接下来谈意识，生活中有不少场景甚至是

仅靠意识而未使用特定技能就最终解决问题的。比如小余直接去楼下买到鱼了，还需要考虑会不会骑车吗？再比如小余让妈妈改做水煮牛肉了，还需要掌握开车的本领吗？

至此方法论说完了，原来买鱼也可以买出方法论来啊。”老师笑着说。

同学们都笑了。

10.3 方法论的应用案例

10.3.1 从我们的这一套流程说起

“好吧，不说故事了，和大家分享点工作中的数据库优化经典案例吧。

某电信营运商生产系统出现故障，短信平台产生大量积压，维护人员跟踪发现，原因是后台短信平台进程调用数据库中某个过程包，而该过程包原先执行返回结果给后台进程，所需时间在 10 秒以内，现在不知是何种原因过程包返回时间居然长达 1 分钟，所以导致短信后台程序处理缓慢了许多，最终造成短信积压。情况紧急，需要立即着手调查，该怎么处理呢？

亲爱的同学们，如何处理你们心中有数吗？如果你真听懂《小余买鱼》，你就应该很有信心来解决这个故障，优化这个系统，成为大家心目中的英雄！不信？让我接着往下讲吧。

该如何优化这个系统呢？好吧，祭出秘密武器：优化方法论！该方法论由买鱼故事演化而来，旗下有两大精髓，又称作八字要领：一套流程、两大法宝！

根据总结的‘一套流程’，我们自然想到，优化的这套流程主要分为诊断和改进优化两个环节，由于现在还不明确问题出在该数据库包的哪个模块，因此进入诊断环节是必要的，然后根据诊断定位出来的具体问题进行具体的优化。”

10.3.1.1 诊断

模拟后台调用该包的时候传入的参数手动执行该包，然后开启 Oracle 的 TRACE 跟踪工具，该过程包执行完毕后关闭跟踪，具体步骤大致如下：

- ① 用 10046 trace 工具开始跟踪：
`alter session set events '10046 trace name context forever,level 12';`
- ② 执行你的数据库包：
`exec pkg_test('abc');`
- ③ 执行包完毕后结束跟踪：
`alter session set events '10046 trace name context off';`
- ④ 10046 trace 工具跟踪完毕后会输出分析结果，类似如下：
`E:\admin\ora10\udump\ora10_ora_4832.trc`
- ⑤ 可格式化后进行分析，类似如下：
`tkprof E:\admin\ora10\udump\ora10_ora_4832.trc d:\10046.txt sys=no sort=prsela,exeela,fchela`

- ⑥ 然后分析 10046.txt 的文件，这里响应时间从大到小展现该包所有 SQL 语句，即可有如下收获：该过程包总共执行了多少 SQL 语句，具体内容是什么，分别开销了多少时长？哪些是开销时长最长的语句？（由于排序过，所以最长的一眼可看出，在最前端。）

咦，这和《小余买鱼 1》中的诊断方法有差别吗？还真是过程细化和找细化项主要矛盾两个动作啊。

接下来分析 10046.txt 文件发现，原来慢的 SQL 是类似如下的两条非常简单的 SQL 语句，分别占用 30 秒和 20 秒，其他所有 SQL 单条执行都只用零点几秒。

语句 1（SQL1 耗时 30 秒）

```
Select count(*) from t1;
```

语句 2（SQL2 耗时 20 秒）

```
Select distinct t1.col1,t1.col2,t2.col3,t2.col4
from t1,t2
where t1.id=t2.id and t1.name='cc'
order by t1.col5;
```

其他 SQL 语句（合计才消耗 0.5 秒）

```
---SQL3      (0.03 秒)
---SQL4      (0.028 秒)
---SQLn.....(0.001 秒)
--略去
```

10.3.1.2 改进优化（首次优化）

“通过分析可以知道，SQL1 和 SQL2 是重点需要改进的 SQL 语句，首先分析 SQL1，该如何改进优化呢？”

如何改进优化？根据小余买鱼的经验，我们知道要先去理解需求，这条 SQL 背后的需求是什么？看上去并不难猜测，就是为了查询 T1 表的记录数，毕竟 `Select count(*) from t1;` 这条语句太简单了。

开始介入分析，通过查看该 SQL 的执行计划可以知道，该表 T1 是进行全表扫描（查看执行计划和 10046 TRACE 一样，是一个非常重要的优化工具）。

该表记录目前有 5 千万条，每次都对全表进行扫描仅为了获取该表的记录。我们的需求是为了得到记录数，是否一定要对全表进行扫描呢？就好比小余要去买鱼，是否一定要去 20 里外的沃尔玛超市呢？

好比小余需要具备了解自己住宅周围是否有地方买鱼这个生活经验一样，这个案例我们也需要多了解点相关知识，啥知识？这里需要读者对 Oracle 索引有比较深刻的理解。

通过全扫描数据表可以获取到该表有多少记录的信息，如果对该表存放序列值的非空字段 SEQ_ID 建一个索引，全扫描该索引，一样可以获取该表有多少记录的信息。看来，获取表记录

数可以有两种方法了。选哪种方法呢？大家知道，索引的大小比表小得多，在更大的范围内遍历更快速还是在小得多的范围内遍历更快速呢？好比买鱼是到远方的沃尔玛快还是到楼下的农贸市场快一样，答案毋庸置疑。

接下来我在 T1 表的 SEQ_ID 这个非空字段上建立一个索引，`Select count(*) from t1;`的语句执行计划从全表扫描转换为索引全扫描。由于该表字段很多，而这个 SEQ_ID 字段又仅占 10 个字节，索引大小仅为表大小的三十分之一。从大范围找答案改为小范围找答案后，意味着我们达成同样的目的，少做了不少事情。如此下来该 SQL 执行速度当然会有大幅度提升，果然，最终执行速度从原来的 30 秒变为 1 秒左右。

看来学习优化还真是和学习买鱼一个道理，顺利优化了 SQL1 就等于完成了工作的一半，继续努力。

接下来进行 SQL2 的调优，这是 T1 和 T2 两表关联的查询，和优化 SQL1 时一样首先开始查看分析 SQL2 语句的执行计划，发现 SQL2 的执行计划也是全表扫描，这里 ‘t1.name=’ 的取值为 cc 的返回仅 10 条记录，而 T1 表记录在 5 千万条左右，T2 表记录在 200 万条左右，需要全扫描这么大的两个表而获取仅有的 10 条记录吗？

这里又要再次利用到索引的原理，SQL1 是利用了索引一般比表小得多的特点，现在又是要利用啥呢？哦，利用索引的快速定位原理。假如我们在 name 列建了一个索引，而现在是利用了索引的快速检索原理。索引有个最大的特点是有序排列，当表记录检索到值为 dc 的记录后，Oracle 就停止遍历了！为啥，因为索引是有序的，dc 是以 d 开头的，后面绝对不可能再出现 c 开头的记录了，因为我们是查询 ‘=cc’ 的值，当然停住了。随时停止检索相比遍历全表，明显是少做事和不做事，效率可以预料会提升不少。

那 SQL2 如何优化，哦，好简单，在 name 列建一个索引就好了。索引在这条 SQL 中因为可以让应用少做事和不做事，最终实现了速度大幅度提升，果然，优化后的执行速度从原来的 20 秒缩减为 1 秒。

到此优化完毕，短信后台进程由原来的每次执行 1 分钟多变为 2 秒多，速度提升了 30 多倍，积压情况大大缓解，系统运行恢复正常。

应该说这次优化总体是很成功的，客户也非常满意。不过我个人心中还是有少许疑惑之处，什么疑问呢？

- ① SQL1 (`Select count(*) from t1`) 为什么要统计条数，得到条数的真正目的是什么？
- ② SQL2 中的 `distinct` 取唯一值是为啥，难道表有重复记录？`distinct` 可是需要排序的。
- ③ SQL2 中的 `order by t1.col5`；排序是 T1 表的 col5 字段，展现字段又没有这个字段，真的需要这个排序吗？”

10.3.1.3 需求与设计（再次优化）

“遇到疑问最重要的就是，去理解需求是什么，真正的需求让我大吃一惊！原来该 SQL1 的语

句在过程包中的代码是类似如下的：

```
begin
select count(*) into v_cnt from t1 ;
if v_cnt>0
then  ...A 逻辑...
else
then  ...B 逻辑...
End;
```

因为这条 `select count(*) from t1` 执行偏慢，所以被 10046 TRACE 追踪到了，但是这个逻辑真的需要这么实现吗？

我来翻译一下这段需求，获取 T1 表的记录数，判断是否大于 0，如果大于 0 走 A 逻辑，否则就走 B 逻辑。因此代码就如上所示来实现了。

真正的需求是这样吗？其实应该是这样的：只要 T1 表里有记录就走 A 逻辑，否则就走 B 逻辑。

两者有区别吗？其实区别还是很大的，前者可是强调获取记录数，我们是不是一定要遍历整个表得出一个记录数才知道是否大于 0？真正需求的理解可以让我们这样实现，只要从 T1 表中成功获取到第一条记录，就可以停止检索了，表示该表有记录了，难道事实不是这样吗？

因此原先的 SQL1 从 `Select count(*) from t1;` 被改造为 `Select count(*) from t1 where rownum=1;` 速度从 1 秒提升到 0.01 秒，几乎可以忽略不计了。这里我们马上联想到《小余买鱼 4》，妈妈真正的需求其实是要做美味的晚餐，站在这个角度，完全可以用现成的原材料来代替非常麻烦的买鱼经历，少做事甚至不做事而快速满足需求，提升效率。

对于 SQL2，我查看了 T2 表，发现真有大量重复记录，怪不得需要用 `distinct` 来排重。我很奇怪为什么会有重复记录，在问明开发设计人员后，我明白了，原来是源头程序有漏洞，导致 T2 表出现大量重复记录，现在所有的应用只要涉及访问 T2 表的，都需要增加 `distinct` 关键字来排重！

天啊，居然还有这回事！还有更神奇的，关于此处的 `order by`，居然是多余的，展现的几个字段的输出根本无须排序，没有这个需求！

接下来思路很简单，就是优化了源头程序（另外一个数据库包），保证插入 T2 表的数据再也不会重复记录，然后再对 T2 表记录进行删除重复记录的操作。

后来 T2 表的记录从原来的 500 万缩减为 20 万，最难得的是，`distinct` 语句可以去掉了（因为不会有重复记录了），`order by` 也去掉了（因为根本没有顺序展现的需求），现在 T2 表不但记录数变小了，排序也都避免了，结果 SQL2 也由原来的 1 秒优化为 0.01 秒，也快到几乎可以忽略不计了。

现在后台短信进程调用该过程包，需要总时间还不足 0.1 秒，比起以前有了成百上千倍的提升，这次优化终于取得了圆满的成功！

最难能可贵的是，很多和这个后台短信进程无关的其他应用也快了！咦，很奇怪吗？其实不奇怪，因为只要是访问 T2 表的应用都少做了两件事，第一是不再需要增加 `distinct` 语句来去重了，避免了排序，第二就是以前访问 T2 表是访问大表，现在是访问小表。”

10.3.1.4 资源利用（花絮）

“同学们还记得我说的 T2 表排除重复记录的事情吧，我当时提供了技术方案给维护人员，方案是新建一张表出来，提取不重复记录，然后把旧表替换了，大致思路如下：

- ① 停应用（这个应该排在第一，防止建表期间有人改数据）
- ② 首先建立新表
`Create table t2_new nologging parallel 12`
`as select distinct * from t2 where;`
- ③ `Rename t2 to t2_bk`
- ④ `Rename t2_new to t2;`
- ⑤ 补上 T2 表的相关索引，并将 T2 表的 logging 属性恢复

我之前提醒过要在业务不繁忙的时候操作，比如凌晨打补丁的时候顺道操作。因为我写了 `parallel 12`，表示要并行用到 12 个 CPU（而当时生产系统仅有 12 个 CPU）。

可惜的是维护人员自作主张，在大白天系统繁忙的时候执行了上述语句，结果导致 12 个 CPU 资源被这条 `create table t2_new` 的语句占据消耗着，引发生产系统短时间的压力繁忙，险些压垮了系统，好在该语句在 5 分钟内结束，未造成严重的影响。

亲爱的读者，这个花絮让你们联想到什么吗？应该是《小余买鱼 3》吧，小余不会合理利用资源，开车去楼下附近农贸市场买鱼，导致要去远方开会的爸爸没车开，迟到了。当然《小余买鱼 2》却是会合理利用资源的典范，爸爸去出差了，要去那么远的地方，不选择开车去就是傻瓜了。如果维护人员按我的要求在凌晨打补丁的时候顺便执行这个语句，不用并行也是不善于利用资源，因为凌晨系统没有什么进程在运行，不会有应用因为 CPU 被占用完而受到影响。”

10.3.2 案例映衬了经典两大法宝

“还记得前面《小余买鱼》故事总结的买鱼方法论中的‘两大法宝’吧，总结的很简单，就是‘意识+技能’。不知道读者注意到没有，这次生产系统的数据库优化不断体现出这两大法宝的作用。

其中意识表现为：①思考该数据库包具体涉及哪几个模块，主要矛盾又在哪个模块？（选择用 10046 TRACE 工具包来实现）；②找到问题所在以后去理解需求，探索是否能少做事完成需求（选择用索引来替代全表扫描，从而减少访问路径）；③去思考需求背后的真正需求（最终将 `select count(*) from t1` 改造为 `select count(*) from t1 where rownum=1`）；④去分析资源如何合理应用（在

系统繁忙时使用并行，占用他人资源，险些酿成大祸)。为什么称之为意识，因为这些并不是具体的知识点技能的使用，更像是一种经验的总结，习惯的培养，所以称之为意识。

那技能呢？其实上述的意识正是和技能紧密联系的。比如 10046 TRACE 诊断工具包的使用和学习；执行计划的查看与分析；理解索引的原理；知道如何使用并行的命令……

Oracle 优化意识的内容大致就在总结的一套流程中，看起来内容不多，不过这些朴素的道理却是优化的核心思想，非常经典！相比 Oracle 优化意识而言，Oracle 技能涉及的范围却有不少，除了前面介绍的一小部分技能外，还有理解 Oracle 体系结构；理解表连接原理；理解 AWR 等性能报表的使用与分析；精通 SQL 和 PL/SQL 编程等。

原来优化方法论和生活中的故事是一样一样的，你们看明白了吗？”

宝典，规范让你少做事

各种合理的规范汇总如图 11-1 所示。

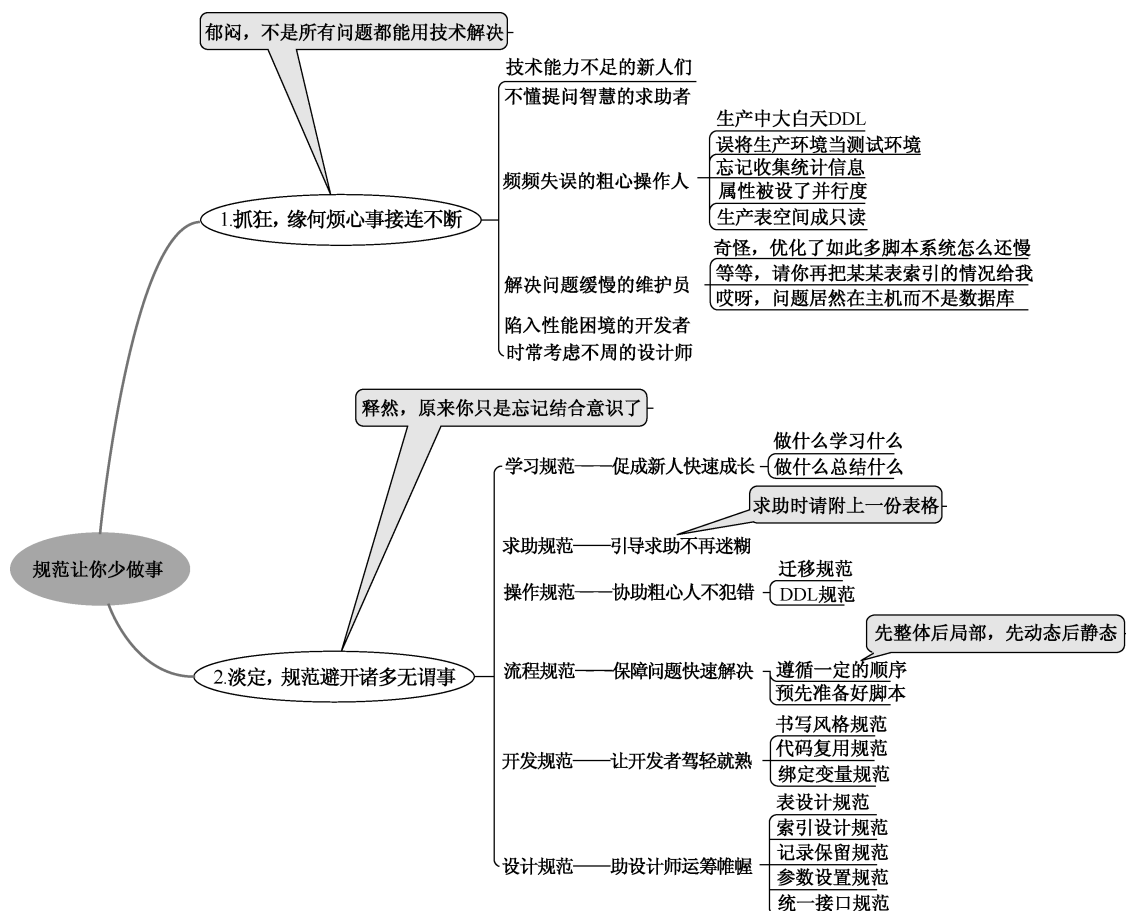


图 11-1 规范让你少做事

11.1 抓狂，为何事总忙不完

“同学们好，今天老师与大家交流规范制定的相关心得体会。大家千万不要小看了规范，俗话说得好，不以规矩，不成方圆。规范能保证工作正确高效地开展，是提高工作效率、少做无谓事的一个非常重要的手段。

老师所在部门的 ITSM 业务遍布全国二十多个省份。在没有制定相应规范的情况下，想同时顶住如此多工程点的压力，绝非一件易事，以我所在的平台项目组为例，整天有来自各个工程点的处理不完的故障、优化不完的代码、审核不完的数据库模型，简直让人抓狂。幸运的是，后来我们想出了有效的应对方案，制定了规范并在部门有效地执行着，所以我们成功地化解了压力，在工作中游刃有余。

在老师描述各类规范宝典之前，请大家先看看未制定相应规范时会遇到哪些问题吧。

11.1.1 技术能力不足的新人们

‘小张，要求你写的那部分模块已经比原计划推迟了 5 天了，什么时候能完成啊？’

‘王经理，不好意思，我这些天都在加班，可是 PL/SQL 编写的技巧我还掌握得不熟练，花费在代码编写调试上的时间太多了。’

‘不是之前已经给你 3 个月时间熟悉业务和学习 PL/SQL 的开发知识了吗？’

‘那段时间我主要精力都在研究备份与恢复，听说这个非常重要。’

‘你不知道自己是要准备做数据库开发吗？’

‘我知道啊。’

‘那你为什么不先学开发方面的知识，比如表和索引的设计、SQL 和 PL/SQL 的编写与优化等等。’

‘我……’

好了，这就是小张的故事，现实中小张比比皆是，其实王经理已经给他规划了学习路线图，指明了先学什么，后学什么。可惜的是，小张就是因为听说备份恢复很重要，没有听进王经理的话。

其实这个案例就是一句话，做什么事情先学什么知识。

如果这成为新人的学习规范，那小张就会受到规范的约束，从而避免了上述事情的发生。

这其实是一个规范问题，关于学习的规范！”

11.1.2 不懂提问智慧的求助者

“接下来说说不懂提问智慧的提问者，哎呀，其实这些案例我已经说过了，大家回想一下老

收获，不止 Oracle

师前面章节《提问，也是智慧的体现》。

该章节中描述的小 A、小 B、小 C、小 D 和小 E 的故事其实都很有代表性，希望能好好体会一下。”

同学们会心地笑了。

“不过如果我们指定了一系列学习的规范，让他们去遵守，那也能极大程度地改善这种情况。后续的章节大家会看到此类规范的。”

11.1.3 产生各种失误的粗心者

“现在我们来讲述产生各种失误的粗心者的系列故事，主人公们都很悲惨，其中还有老师自己，都是一部催人泪下的血泪史啊。”

不少同学听了忍不住笑出声来。

11.1.3.1 啊，小黄的 DDL 惹祸

“大家别笑，如果这些事情发生在你们身上，你们就笑不出来了。

某次生产某系统忽然出现故障，该平台原本可以在 1 秒内正常返回的话费查询短信忽然产生延迟，很多客户发出短信后 20 分钟收不到返回，引发了大量的客户投诉。

在排除问题的过程中，发现故障是由于处理短信的后台程序忽然 HANG 住无法执行。HANG 住的原因是因为该程序处理的 T 表被锁住了，由于锁资源一直未释放导致程序对该表的 DML 操作无法成功，进入排队等待的队列中。

接下来继续分析，为什么表会被锁住，甚至是全表锁，这是怎么回事呢？经过进一步的排除发现，原来是类似如下一条语句将 T 表锁住了：

```
create index idx_id on t(id);
```

这是一个非常普通的建索引语句，怎么把生产搞出故障来了？其实原因很简单，在该表建索引时产生了锁，直到建完后才释放。所以在建索引期间，后台程序处理该表时产生了锁等待。为什么会产生锁，其实也很简单，因为索引是有序的，建索引过程中要把该索引列的所有数据都排序并存入数据块中，形成索引块。如果没有对该表加上锁，那记录不断更新，排序动作何时可以结束，索引又如何建好呢？

是谁干的好事啊？原来是小黄，他发现如果在 T 表的 ID 列上有一个索引，可以让他的模块的对应 SQL 跑得更快，当即就在大白天执行了这个操作，然后就酿出大祸了。这个操作显然应该是放在晚上非业务高峰期时完成，如果这样，问题就可以避免了。

这其实就是一个规范的问题，关于操作的规范！”

11.1.3.2 惨，老师登错环境了

“这是发生在老师自己身上的一个悲惨故事，多年前的某个晚上，我对电信某系统进行应用补丁升级，操作流程中我需要先后多次在生产系统和测试环境中登录，同时完成两个环境的系列操作。

或许是因为工作太疲惫了，我居然在生产环境中执行了测试环境的脚本，把生产环境中的某些表的结构和记录都更新成和测试环境一样。

这是一个很可怕的事情，生产中的应用程序瞬间就出问题了。当我意识到这一点时，已经太迟了。虽然是严冬季节，但是我已经满头大汗了，接下去做的操作就是恢复刚才错误的操作，可是这些脚本涉及的表众多，还有更改结构的 DDL 语句，该如何恢复？

情急之时想到备份与恢复，可是之前未针对这系列被操作的表做备份，又如何恢复呢？如果只是 DML 语句，还可以考虑用闪回来操作，可是有部分被 DML 的表是有 LONG 字段的，并且还涉及 DDL 语句，靠闪回也是不行的。当时瞬间觉得万念俱灰，跳楼的心都有了。

最后冷静下来，找到该生产环境的相关平台人员，确认了其备份的机制，于当天晚上做了全库的恢复，将数据库恢复到操作失误前的那个时刻点。所幸的是由于凌晨 1 点开始打补丁时停了所有应用，1 点到当前 2 点之间系统处于应用程序停止运行状态，所以恢复到 1 点这个时间点，数据库并不会出现不一致的情景。

恢复完毕后，再启动应用程序，几个凌晨被从家里叫来的测试组的同事紧张地投入到测试工作中，最终证实系统确实是正常的，大家这才松了一口气。不知不觉已是凌晨 5 点了。

这是一件惊心动魄的难忘之事，会让我记忆一辈子，现在我在进行任何相关操作时，都会提醒自己，请弄清楚你操作的环境。其实这次的故障要避免也很简单，只要你保证操作生产环境时，关闭测试环境的登录界面；操作测试环境时，关闭生产环境的登录界面，那么无论你怎么疲惫，都不容易出错了。

不用说了，还是规范的问题！”

11.1.3.3 糟，小罗忘操作……

“某天早上 6 点忽然接到公司值班人员紧急求救电话，说发现某生产系统出现严重故障，前后台应用程序全部处理失败。

有这等事，惊得我甚至顾不上刷牙洗脸就匆匆出发赶到公司，到了现场根据日志一看，原来是前后台程序在更新数据库表时失败，再仔细一看，原来是某表空间被设置为只读了，真是一件让我万分吃惊的事。

接下来的事情很简单，就是将该表空间的只读属性设置为读写属性，故障瞬间解决，系统恢复正常。

到了追究责任的时候了，经调查发现，昨天晚上至当天凌晨小罗操作了数据库迁移的相关事情，用的是传输表空间的方式。大致步骤如下：

- ① 将旧库表空间设置为只读。
- ② 将新库表空间设置为只读。
- ③ 将旧库表空间的数据文件复制到新库所在的主机上。
- ④ 执行系列转换命令。
- ⑤ 将旧库表空间恢复为读写。
- ⑥ **将新库表空间恢复为读写。**

问题出在哪里呢，同学们？”老师停下来问大家。

“是不是第 6 步？小罗第 6 步忘记操作就结束了迁移动作。”晶晶问。

“没错！就是第 6 步给忽略了。后来我找到小罗，刚和他说到表空间只读，他就猛然醒悟过来，惊叫一声，糟，我忘记把表空间恢复为读写了，不好意思，我太粗心了！”

小罗有着多年的工作经验，关于传输表空间的迁移也做过多次，这次为什么会出现这个问题呢？还是因为他缺少了流程规范，如果小罗的操作有流程步骤同时还有项目经理等其他人的步骤确认校验过程，这种情况会发生吗？”老师停下来问大家。

“不会，有流程步骤加上专人审核确认，那肯定好多了！”小莲大声回答。

“关于步骤缺失导致生产出故障的情况不胜枚举，老师再举一个例子，也是在一次迁移工作完成后，系统忽然遭遇到了巨大的性能瓶颈，由于对生产影响巨大，我正在机场准备登机，临时放弃登机，赶回现场。

经过排查，很快发现问题所在，系统存在大量的 PX Deq 的等待，这很显然是属于表或索引被设置了并行的属性导致的，进一步查询果真得到确认，数据库中大量的表被设置为并行度为 8 的属性。由于表的属性被设置为并行，导致程序动辄产生大量并行，主机 CPU 资源本身就有限，互相争用，最终导致性能一塌糊涂。

该如何处理，很简单，将表的并行度消除后，问题立即解决，系统瞬间恢复正常。

又到了追究责任的时候了。

原来是小赵做的如下大表迁移操作。

- ①

```
CRETAE TABLE T1 NOLOGGING AS SELECT * FROM T1@dblink parallel 8;
CRETAE TABLE T2 NOLOGGING AS SELECT * FROM T2@dblink parallel 8;
CRETAE TABLE T3 NOLOGGING AS SELECT * FROM T3@dblink parallel 8;
.....
```
- ② 将表的属性设置为 LOGGING:

```
ALTER TABLE T1 LOGGING;
ALTER TABLE T2 LOGGING;
ALTER TABLE T3 LOGGING;
.....
```
- ③ **将表的并行度消除：**

```
ALTER TABLE T1 NOPARALLEL;  
ALTER TABLE T2 NOPARALLEL;  
ALTER TABLE T3 NOPARALLEL;  
.....
```

问题出在哪里呢，同学们？”老师再次停下来问大家。

“小赵把第 3 步操作给忘记了吧。”又是晶晶回答。

“说得很好，正是此处操作给忽略了。之前我们已经总结过了，有流程规范加上专人审核确认，这些问题完全可以避免！

粗心只是一种表象，规范缺失才是真正需要改进的！”

11.1.4 解决问题缓慢的技术员

11.1.4.1 优化效率低下的小高

“某次河南某生产系统出现性能瓶颈，前台相关操作人员投诉系统前台界面操作比之前慢很多，要求我帮忙分析优化。

正好我因为家中急事准备休假一周，就将此事移交给了同事小高来解决。

一周后我休假回来，小高沮丧地告诉我河南的问题还没有解决好，这一周以来他已经和相关配合人员做了不少优化改进，有收到一点效果，不过总体还是运行比较缓慢，不知该如何进一步优化。

我曾建议小高着手解决问题前先建一个 QQ 讨论组，让他把相关配合人员加入讨论组，这样更方便交流和解决问题。不过看到这个 QQ 讨论小组的聊天记录后，我明白了为什么小高花了一周时间，都没能解决好问题。

在这个讨论群里，我没能看到我所关心的讨论话题：

- ① 通过聊天记录我看不出系统是每时每刻都慢，还是只是某个时间段运行缓慢。
- ② 通过聊天记录我也看不出是前台系统的所有菜单都运行缓慢还是部分菜单缓慢。
- ③ 通过聊天记录我也看不到小高在了解前台运行异常之前系统有无做过何操作。

这三个问题是我在解决问题前最想了解，因为往往能快速发现问题的矛盾所在，可惜小高没有去关心，求助者也没有反馈。

通过聊天记录，我看到小高做了如下几件收集准备工作：

- ① 收集过 AWR 报表，不过他收集的是基于系统运行最近一周以来运行情况的报表。
- ② 找相关人员配合，不断让现场配合人员执行他提供的各类语句来了解数据库和主机的参数情况。

- ③ 找出了不少认为运行比较慢的 SQL 语句，并收集了执行计划来分析。
- ④ 找相关人员配合，不断让现场配合人员执行他提供的查询表和索引情况的语句。从而去了解这些 SQL 涉及的表和索引的情况。

此外还看到小高提出的不少建议：

- ① 给 XX 表增加索引。
- ② XX 表的 XX 索引失效，建议重建 XX 索引。
- ③ XX 表的统计信息收集有误，建议全局临时表不要收集统计信息。

从讨论组里的人回复交互的情况来看，现场人员配合得非常积极，基本上都在小高提出改造建议的当天晚上（业务高峰期过后）完成了索引、参数等相关改造工作。

虽然大家讨论很热烈，配合很积极，可惜结果不理想，一周下来问题没有得到有效的解决。

11.1.4.2 为何老师能快速解决

随后优化河南系统的事就交给我了，我首先了解到了我最关注的三个问题：

- ① 系统不是每时每刻都慢，只是每天早上 10 点到 11 点以及下午 3 点到 4 点期间慢。
- ② 出现性能问题时，不只是某个菜单运行缓慢，是全部模块都特别慢。
- ③ 出现问题前系统有做过一次打补丁的操作，在数据库层面部署了一个定时任务，用来定时收集某些采集信息。

了解到这些后，我不像小高那样收集一整周的 awr 性能报表，也不是收集一整天的性能报表，而是只收集 10 点到 11 点及下午 3 点到 4 点这两个时间段的报表。通过分析后我发现这两段时间内，数据库存在严重的 latch 相关等待。进一步通过 ash 报表后明确这些等待是由某些 SQL 引发的。

接下来我立即怀疑和前面说的定时任务有关，我了解了定时任务的执行规律，早上 10 点开始运行，每 1 分钟执行一次，11 点后结束。下午 3 点开始运行，每 1 分钟执行一次，下午 4 点结束。

现在一切问题都明白，问题已经定位清楚了。分析这些 SQL 的执行计划，发现这些 SQL 涉及的表比较大，有数百万条，返回的查询或者更新每次不过数条，却用的是全表扫描。比如某 SQL 的写法如下：

```
select sum(a.store_size), TRUNC(a.generate_time) generateTime
  from idep_goods_log a, idep_goods b, idep_product c
 where (a.goods_id = b.goods_id(+) and b.goods_id is not null)
       and (b.prod_id = c.prod_id(+) and c.prod_id is not null)
       and c.prod_id = 123
```

```
and TRUNC(a.generate_time) >= TO_DATE('2012-08-02', 'YYYY-MM-DD')
and TRUNC(a.generate_time) <= TO_DATE('2012-08-03', 'YYYY-MM-DD')
group by TRUNC(a.generate_time)
order by TRUNC(a.generate_time)
```

这个语句只是 JOB 模块的一部分而已，JOB 要求在 1 分钟内完成，而仅仅这个部分就需要执行近 5 分钟。由于每过一分钟系统不断启动新的 JOB 程序，重复执行该 SQL，最终导致在 10 点到 11 点那个时段，系统的所有 CPU 资源都被这个定时任务给耗尽了，这个优化很简单，因为在 generate_time 列本来就有索引，但是上面加粗显示的两行语句的写法显然是用不到索引的，只需将这两行语句改为如下，即可用到索引：

```
and a.generate_time >= TO_DATE('2010-08-02', 'YYYY-MM-DD')
and a.generate_time < TO_DATE('2010-08-03', 'YYYY-MM-DD')+1
```

由于查询只是限于一天的记录，返回记录很少，所以这个语句在用到索引后仅 0.1 秒就查询出结果了。河南的问题很快就解决了。

这次优化我仅仅花费了不到 20 分钟的时间，而小高花费了一周时间，却根本没了解到有这个定时任务，当我告诉他我的优化经过后，他有些沮丧。

其实无法有效解决问题的真正原因在于小高不善于发现主要矛盾。小高的报表是一周以来的报表，从一周以来的情况来看，刚才的 SQL 语句运行时间 5 分钟，频率一小时 60 次，一天就 120 次。无论从时长还是频率来说，在一周的报表里都排不上号，所以小高根本没有发现这个问题。

但是如果这个报表体现在 10 点到 11 点这个时间段，那就是报表中的头号通缉犯了。

我并不是神，为什么我知道看 10 点到 11 点这时段的报表？因为我想抓住主要矛盾，我问了相关人员，了解到了关键问题所在。

这是一个很经典的例子，希望大家好好体会一下，为了加深大家的印象，我最后给大家举一个生活中的例子。比如这个城市交通情况，如果白天 12 小时下来，交通拥挤的时间共计 50 分钟，那我们不好判断这个城市交通有多拥挤。但是如果这 50 分钟里有 40 分钟是在早上 8 点到 9 点的时间段里，那就可以说明这个城市早高峰的交通拥挤程度非常严重！

这和小高的故事是一样的，抓住主要矛盾，快速解决问题。其实生活和工作是没什么本质区别的。

小高的失误其实主要在于解决问题的相关经验不足，但是所有的人都是从经验不足成长起来的，所以我们不能责怪小高。但是有一种方法却可以快速弥补经验不足的缺陷，那就是规范。我在工作中总结解决故障问题的系列流程规范，从如何询问故障，到如何抓住主要矛盾，到如何获取具体的相关信息，都有详细的说明。然后我将自己总结的规范流程在部门及公司层面进行培训，让大家快速理解和掌握。

现在的小高，已经不再是从前的小高了，从规范定制至今短短的三个月，小高快速解决问题

的能力已经得到了所有同事和领导的认可，他也非常开心。

哇，原来经验不足也可以用规范来弥补！”

11.1.5 陷入种种困境的开发者

“这里给大家说一些陷入种种困境的开发者的小故事，希望大家能从他们身上受到启发。首先听听开发人员小郑与领导的一段对话。

11.1.5.1 超长 SQL 使小郑烦恼

‘小郑，让你根据需求调整一段代码，怎么两天还没好啊，有啥困难？’

‘对不起姚总，因为没法在原先 SQL 的基础上做调整，所以我在重写。’

‘重写？不就是在原来的基础上增加一个小需求吗，为何要重写？’

‘因为原先的需求是这样实现的，您看，就这一条 SQL，挺长的，新的逻辑不知道怎么加进去。’

‘哇，这么长的 SQL，210 行！’

‘是啊，我当时就想直接写成一条 SQL，用 Java 直接封装了，调用方便，现在有些无从下手，不知该如何扩展下去。’

‘你确实是要考虑重写了，不只是因为无法扩展了，这么复杂的需求写成一条 SQL，你的异常怎么写，你的注释又如何写呢？交给新人维护，新人又如何接手呢？’

‘是啊，姚总，现在这个语句还有一个问题，就是在某些工程点跑得很快，在某些工程点跑得很慢。’

‘那是因为语句太过于复杂，导致 Oracle 的执行计划也非常复杂，不同的工程点环境有差异，比如表的数据量、统计信息以及数据库参数都会影响执行计划。

你赶紧分开重写了，可以考虑在过程包中实现，分步进行，尽量多写一些注释，同时要考虑异常情况和日志跟踪的设计，以方便将来调试。’

‘哦，我马上按您的意思去做，多谢！’

这是非常典型的一个案例，写超长 SQL 的人员非常多，而依我的经验来看，超过 100 行的 SQL 基本上都是不合适的，所以我制定了一个规范，长度超过 100 行的 SQL 都需要考虑改造。

规范真的很重要，我所在的各个项目组中，很少有超长的 SQL，因为规范已经制约了。

接下来我们再听听小叶的故事，也从一段对话开始了解。

11.1.5.2 缺少注释让小叶沮丧

‘姚总，刘工的代码我读不懂什么意思啊？’

‘读不懂，为什么啊小叶？’

‘因为都没注释啊，我不知道这些逻辑是啥意思，所以我没办法在刘工的代码上做修改。’

‘那你找一下刘工吧。’

‘我找过刘工了，他说时间太久了，他忘记是什么逻辑了。’

‘不可能吧，小叶，你把代码拿来给我看看。’

‘您看。’

‘哇，别说你了，我也看不懂，这么长的代码，居然没有看到什么注释，怪不得小刘本人也记不起来了。’

‘那咋办啊姚总？’

‘郁闷，交给我晚上加班研究研究，写一点注释进去后再交给你吧……’

同学们，这又是一个经典案例，注释在开发中是非常重要的，开发人员忘记自己代码逻辑的情况是非常常见的，如果要避免上述情况的发生，除了宣贯外，还必须将其写进开发规范中，进行约束和监督。

规范真的很重要！我们的开发规范中，明确规定注释不得少于代码的十分之一，在指定规范后，上述浪费时间的烦恼事在部门消失了。”

11.1.6 总是考虑不全的设计者

“下面的故事和设计人员有关。请听老师描述一段经典的对话。

11.1.6.1 未提前规划的王工

‘王工，XX 系统的 T1 表保存了近 2 年的记录，共计 2 亿多行，大小有 20GB 这么大。现在 T1 表的操作无论是更新还是查询都很慢。我们和需求方确认后敲定该表只要保留最近 1 个月记录即可，怎么清理最快？’

‘小张，T1 表记录会涨这么快，真没想到。另外你确定只需保留最近 1 个月的记录吗？’

‘是的，该表每月记录平均有 1 千多万条。关于记录仅需保留 1 个月是陈主任确认的。’

‘这样啊，早知道我就会对 T1 表建分区，这样检索记录在指定的分区中完成，即便表非常大也可以缓和你前面提到的性能问题。另外你说的记录只需要保留最近 1 个月，那更应该用分区了，分区删除速度快。’

不好意思，主要是我不了解这个模块的数据流向情况，后续我们找一个时间将 T1 表改造成分区表吧，不过操作要停应用进行。哦，对了，应用程序也要调整……’

‘不是吧，王工，这么麻烦啊。’

‘是啊，我也头疼啊。’

这里的问题很显然是因为在设计之初，对业务的规模和特征缺乏了解，对数据的增长以及保

留情况缺少规划导致的。当然，这和工作经验还是关系紧密的，但是如果在设计之初，有一个设计规范明确让你去了解这些情况，且需要专人跟进和评审，就完全能有效地避免这次失误。

规范，能帮上你大忙的，依然是规范。”老师说到到这里忽然停下来，留点时间给同学们回味。

11.1.6.2 不了解特性的刘工

“接下来，我们再来看一段对话。

‘刘工，今天我们发现系统存在严重的锁等待，后来定位到是 T 表的 FLAG 字段被建了位图索引，正是这个索引引起的，听说这个索引是你建的，要考虑赶紧删了。’

‘梁老师，建这个索引没问题吧，这个 FLAG 列就三个取值 0、1、2，分别表示未处理、处理中、已成功。取值如此少，这列大量重复，符合建位图索引的情况啊。’

‘问题是你这个列是要频繁更新的啊，位图索引所在的列一旦被更新，就会把这列的大部分记录都给锁住了，比如你将 0 改为 1，那你就把 T 表 FLAG 列中取值为 0 和 1 的所有行都锁住了。’

‘梁老师，不是吧，真会这样？我没想到啊！’

‘是的，位图索引应用的场合除了索引列是尽量重复外，还要保证更新不怎么频繁，两个条件缺一不可。’

‘哦，学习了，看来我要修改表结构，调整数据库模型了，谢谢！’

类似的问题还有主外键关联时，要考虑在外键上建索引，从而避免在更新主外键表时产生的锁等待。这些其实都是基于 Oracle 数据库开发时的一些基本常识，应该写进规范里，让设计人员明白。

规范可以弥补知识面的不足，从而让你少犯无谓的错误！

至此，老师把因为不规范带来的各种痛苦的案例和大家分享完了，大家是不是觉得听起来有些让人心存畏惧？”老师笑着问。

台下同学纷纷点头。

“老师分享这么多案例，希望大家能牢记在心，引以为戒。不过大家大可不必因此在数据库的相关学习工作中畏手畏脚，如果在我们的共同努力下能制定并完善一系列规范，严格按照规范来进行针对数据库的学习工作，那就基本上可以保证万无一失。

下面我们就开始进行规范的探讨，主要从学习、求助、操作、故障处理流程、开发、设计这 6 个角度进行探讨，其中学习、求助操作规范我就不详细展开，又由于数据库的操作种类繁多，操作规范也无法一一详细展开。不过我不细说绝对不代表不重要，而是抛砖引玉，仅展现流程图给大家去学习和认识。大家一定要有这些意识，达成共识并学会自我总结完善，最后严格遵守。

老师详细展开的重心在故障处理流程、开发、设计这 3 个规范上，这 3 个规范主要是针对 Oracle 数据库的规范。这是老师尝遍了酸甜苦辣，汇集多年的心血总结而成的，并且在公司被广泛应用，取得了极大的成功。

这几个规范最大的亮点之一在于有大量的脚本可以验证。比如大家都在谈设计一定要重视，可是实际情况是，很多产品没有按照规范却已经设计好了，在生产系统中运行了。很多没注重开发规范的代码也在生产系统中悠闲地跑着。

这时不要紧，我们可以利用规范定制中的检验脚本来检查生产系统的运行情况，找出问题后，再进行整改，这样规范就永远不会停留在口头上而无人执行了。

下面，让我们一起进入老师本次系列课程的最后一个章节，激动人心的规范分享。大家切记，这个规范永远都不是最好的，也不是最全的，需要大家不断去补充，去改进，去完善。我们要的是有规范意识，尽量利用规范来避免错误，少做无谓的事情！”

11.2 淡定，规范少做无谓事

“同学们好，请看图 11-2，这是关于数据库 6 大领域规范的总体流程图。

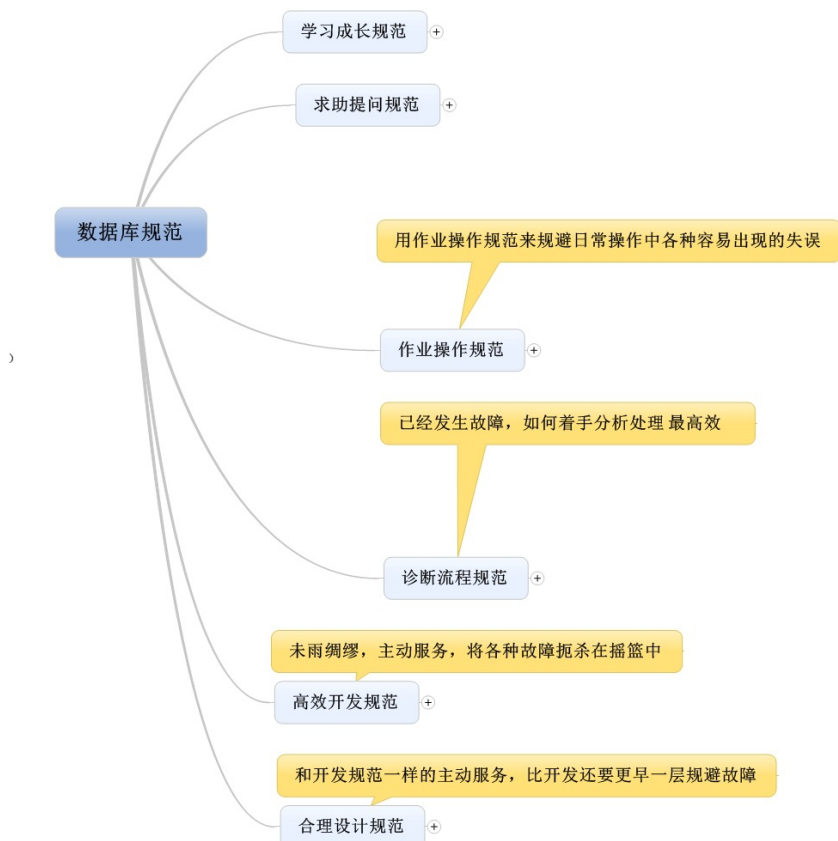


图 11-2 数据库规范

老师分享的重心在后三大规范，前三大规范我就仅以展现流程图的方式和大家交流，暂不详细展开，首先从学习规范看起。”老师停下来喝了口水。

11.2.1 学习规范——促成新人快速成长

学习成长规范如图 11-3 所示。

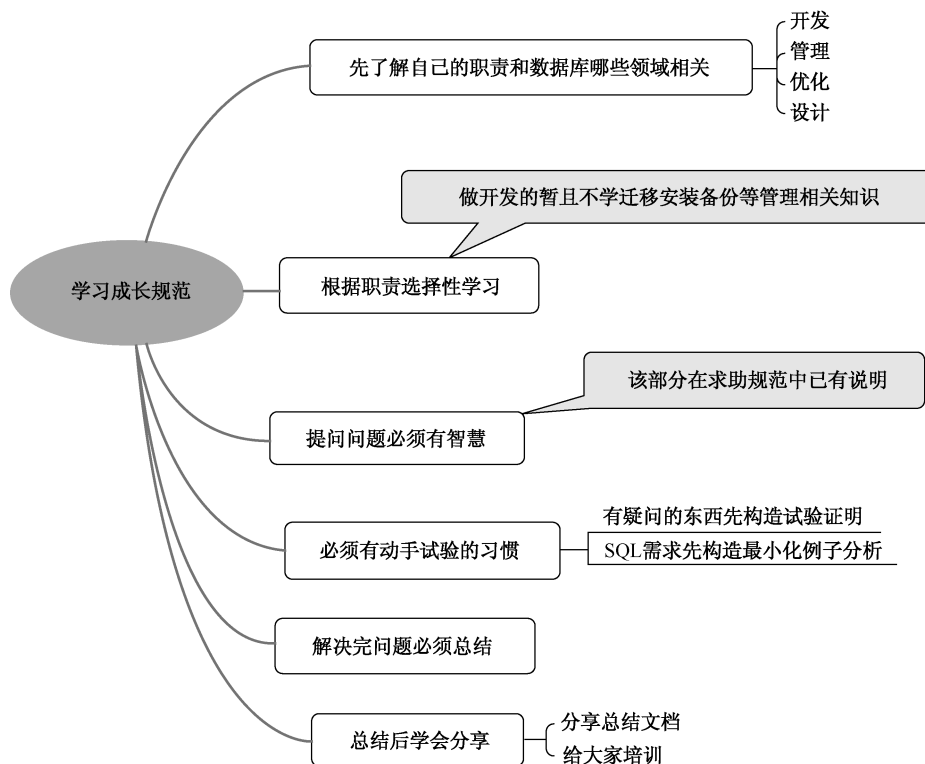


图 11-3 学习成长规范

11.2.2 求助规范——引导求助不再迷糊

求助规范如图 11-4 所示。

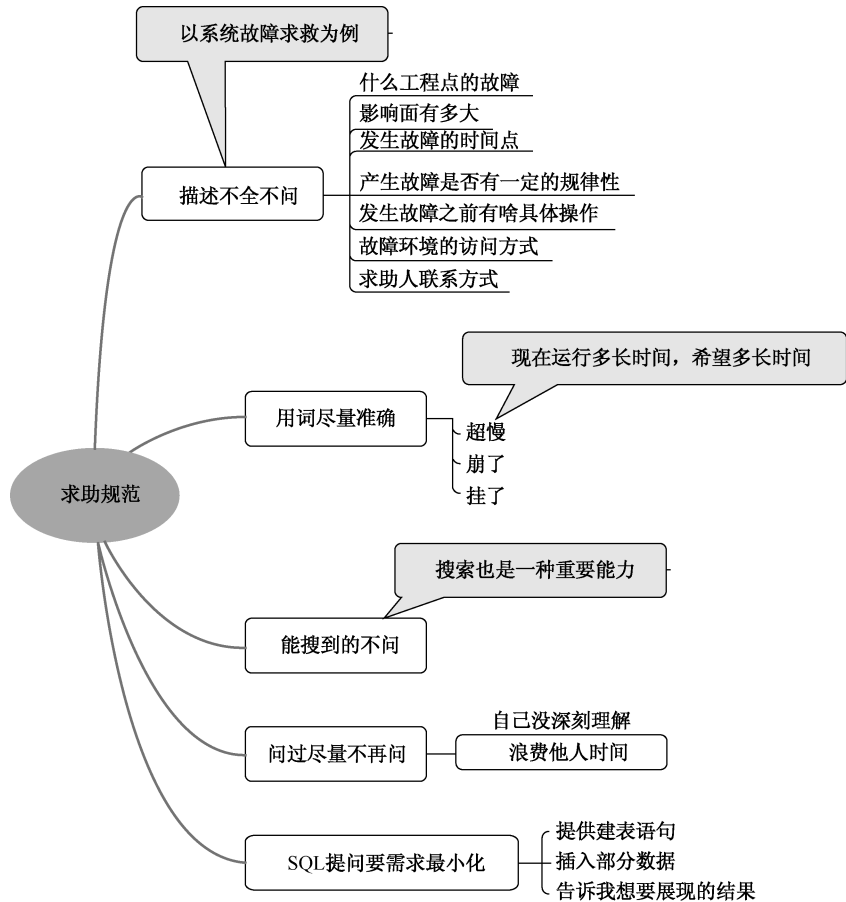


图 11-4 求助规范

11.2.3 操作规范——协助粗心者不犯错

操作规范如图 11-5 所示。

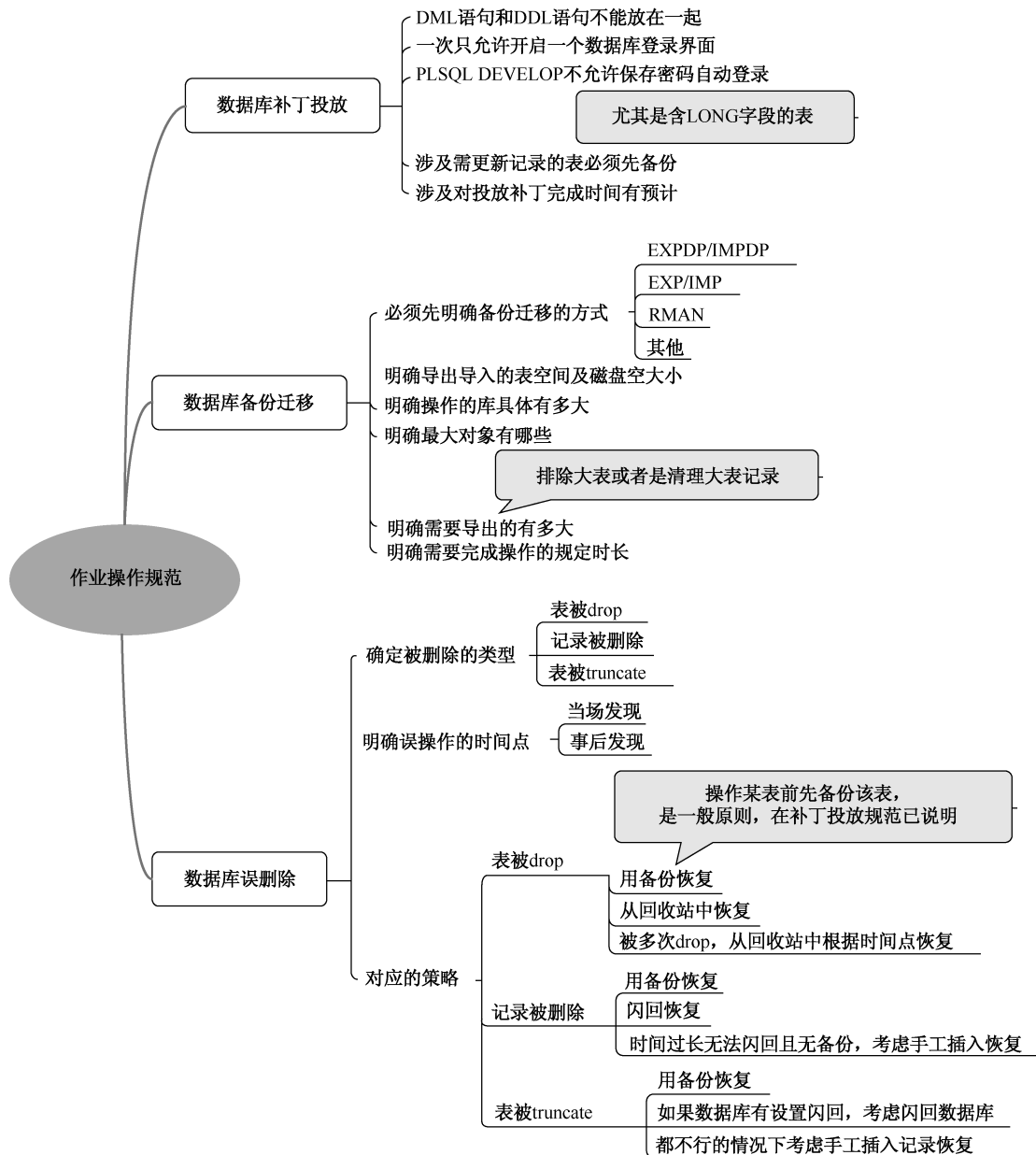


图 11-5 操作规范

11.2.4 流程规范——保障问题快速解决

问题解决的流程规范如图 11-6 所示。

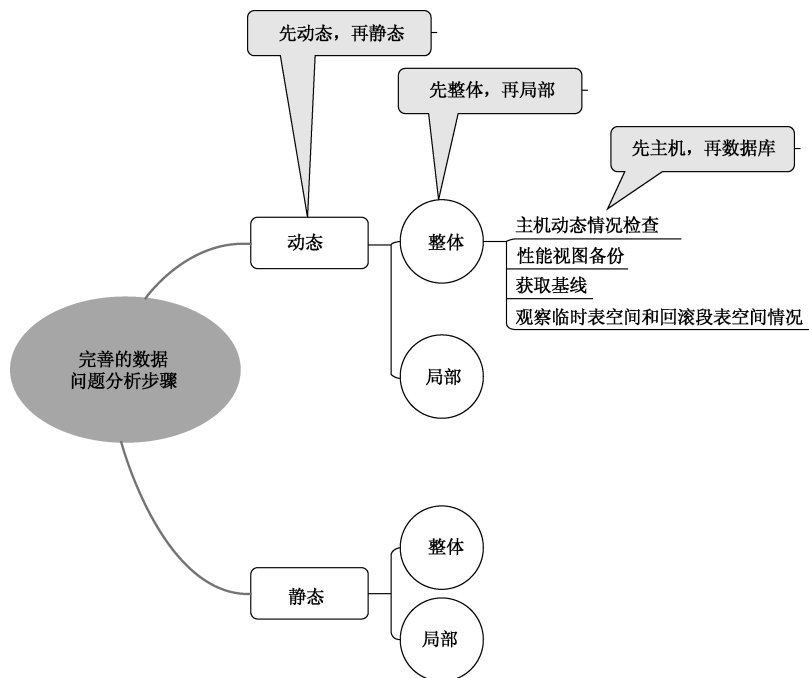


图 11-6 数据库问题解决

11.2.4.1 动态整体

动态整体问题的解决流程如图 11-7 所示。

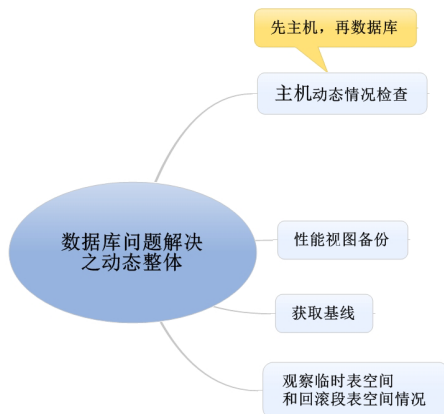


图 11-7 数据库问题解决之动态整体

收获，不止 Oracle

1. 主机动态情况检查

主机情况检查(数据库出问题，主机是首先要查看的，皮之不存，毛将焉附)

```
uname -a
```

top 或者 topas ---检查主机 CPU 等使用情况（重点关注时间最长的，同时也注意观察主机的内存的物理大小和 CPU 的个数）

```
vmstat 1 10
```

```
ps -ef |grep ora |wc -l
```

```
ps -ef |grep ora|grep LOCAL |wc -l
```

2. 性能视图备份

考虑备份数据库性能视图(最好是在新建的非生产使用的单独用户下操作，比如 ljb 用户)

--此外在数据库需要重启时，更应考虑备份这些视图

```
sqlplus "/ as sysdba"
```

```
create user ljb identified by ljb;
```

```
grant dba to ljb;
```

```
connect ljb/ljb
```

create table diag_session_&yyyymmdd_seq_area	nologging as select	* from gv\$session;
create table diag_ses_wait_&yyyymmdd_seq_area	nologging as select	* from gv\$session_wait;
create table diag_process_&yyyymmdd_seq_area	nologging as select	* from gv\$process;
create table diag_sql_&yyyymmdd_seq_area	nologging as select	* from gv\$sql;
create table diag_sqlarea_&yyyymmdd_seq_area	nologging as select	* from gv\$sqlarea;
create table diag_sql_plan_&yyyymmdd_seq_area	nologging as select	* from gv\$sql_plan; --耗性能
create table diag_lock_&yyyymmdd_seq_area	nologging as select	* from gv\$lock;
create table diag_loc_obj_&yyyymmdd_seq_area	nologging as select	* from gv\$locked_object;
create table diag_access_&yyyymmdd_seq_area	nologging as select	* from gv\$access;
create table diag_latch_&yyyymmdd_seq_area	nologging as select	* from gv\$latch;
create table diag_latch_chil_&yyyymmdd_seq_area	nologging as select	* from gv\$latch_children;
create table diag_Libcache_&yyyymmdd_seq_area	nologging as select	* from gv_\$Librarycache;
create table diag_rowcache_&yyyymmdd_seq_area	nologging as select	* from gv_\$rowcache;
create table diag_sort_seg_&yyyymmdd_seq_area	nologging as select	* from gv\$sort_segment;
create table diag_sort_usa_&yyyymmdd_seq_area	nologging as select	* from gv\$sort_usage;
create table diag_log_hist_&yyyymmdd_seq_area	nologging as select	* from gv\$log_history;
create table diag_log_&yyyymmdd_seq_area	nologging as select	* from gv\$log;
create table diag_logfi_&yyyymmdd_seq_area	nologging as select	* from gv\$logfile;
create table diag_transa_&yyyymmdd_seq_area	nologging as select	* from gv\$transaction;
create table diag_param_&yyyymmdd_seq_area	nologging as select	* from gv\$parameter;
create table diag_ses_lon_&yyyymmdd_seq_area	nologging as select	* from gv\$session_longops;
create table diag_bh_&yyyymmdd_seq_area	nologging as select	* from gv\$bh;
create table diag_filest_&yyyymmdd_seq_area	nologging as select	* from gv\$filestat;
create table diag_segst_&yyyymmdd_seq_area	nologging as select	* from gv\$segstat;
create table diag_temst_&yyyymmdd_seq_area	nologging as select	* from gv\$tempstat;
create table diag_datfi_&yyyymmdd_seq_area	nologging as select	* from gv\$datafile;

```
create table diag_tmpfi_&yyyymmdd_seq_area      nologging as select      * from gv$tempfile;
create table diag_opencur_&yyyymmdd_seq_area      nologging as select      * from gv$open_cursors;
```

3. 获取基线

当系统觉得有问题时，可以考虑立即取一个断点基线，作为 AWR 报表的一个断点。

```
sqlplus "/ as sysdba"
exec dbms_workload_repository.create_snapshot();
```

4. 观察临时表空间和回滚段表空间情况

--查谁占用了 undo 表空间

```
select r.name 回滚段名,
       rsize / 1024 / 1024 / 1024 "rsize(g)",
       s.sid,
       s.serial#,
       s.username 用户名,
       s.status,
       s.sql_hash_value,
       s.sql_address,
       s.machine,
       s.module,
       substr(s.program, 1, 78) 操作程序,
       r.usn,
       hwmsize / 1024 / 1024 / 1024,
       shrinks,
       xacts
from sys.v_$session      s,
     sys.v_$transaction t,
     sys.v_$rollname      r,
     v_$rollstat          rs
where t.addr = s.taddr
     and t.xidusn = r.usn
     and r.usn = rs.usn
order by rsize desc;
```

--查谁占用了 temp 表空间

```
select t.blocks * 16 / 1024 / 1024,
       s.username,
       s.schemaname,
       t.tablespace,
       t.segtype,
       t.extents,
       s.program,
```


收获，不止 Oracle

```
s.osuser,  
s.terminal,  
s.sid,  
s.serial#  
from v$sort_usage t, v$session s  
where t.session_addr = s.saddr;  
  
--还可查到具体 SQL  
select sql.sql_id,  
       t.blocks * 16 / 1024 / 1024,  
       s.username,  
       s.schemaname,  
       t.tablespace,  
       t.segtype,  
       t.extents,  
       s.program,  
       s.osuser,  
       s.terminal,  
       s.sid,  
       s.serial#,  
       sql.sql_text  
from v$sort_usage t, v$session s, v$sql sql  
where t.session_addr = s.saddr  
      and t.sqladdr = sql.address  
      and t.sqlhash = sql.hash_value;
```

11.2.4.2 动态局部

动态局部问题的解决流程如图 11-8 所示。

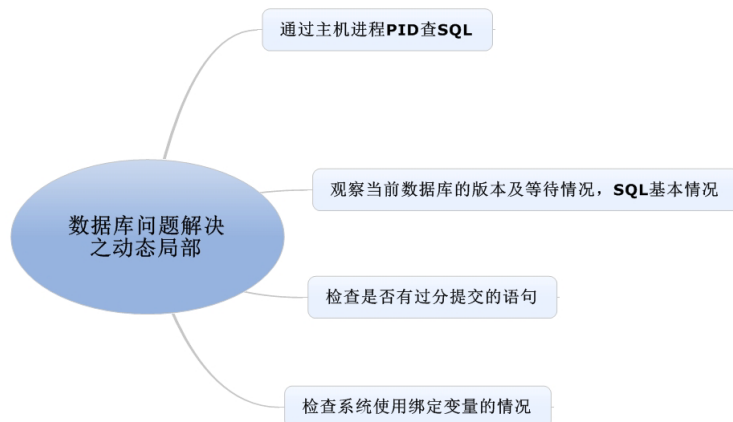


图 11-8 数据库问题解决之动态局部

1. 通过主机进程 PID 查 SQL

通过主机进程 PID 查 SQL（这个步骤和之前的 top 命令紧密相连，就是为了直接分析这些耗 CPU 的进程和哪些 SQL 有关系）本来可以用如下方法来查，但是系统出现问题时，一般不容易查出来，太慢（有时用 ordered 或者 no_merge 的 HINT 有效，有时无效）

```
select /*+ ordered */
  sql_text
  from v$sqltext a
 where (a.hash_value, a.address) in
        (select decode(sql_hash_value, 0, prev_hash_value, sql_hash_value),
              decode(sql_hash_value, 0, prev_sql_addr, sql_address)
         from v$session b
        where b.paddr =
              (select addr from v$process c where c.spid = '&pid'))
 order by piece asc;
```

我们一般可采用如下三步骤来完成（可避免性能问题，返回结果会快）

---步骤 1（关键是获取到 addr 地址）

```
select spid,
       addr,
       t.pga_used_mem,
       t.pga_alloc_mem,
       t.pga_freeable_mem,
       t.pga_max_mem
  from v$process t
 where spid = '1667438';
```

---步骤 2（根据 addr 地址，关键得到 sql_id）

```
select t.sid,
       t.program,
       t.machine,
       t.logon_time,
       t.wait_class,
       t.wait_time,
       t.seconds_in_wait,
       t.event,
       t.sql_id,
       t.prev_sql_id
  from v$session t
 where paddr = '070000018fbf9210';
```

收获，不止 Oracle

---步骤 3（根据 sql_id 知道是什么 SQL）

```
select t.sql_id,  
       t.sql_text,  
       t.executions,  
       t.first_load_time,  
       t.last_load_time,  
       t.buffer_gets,  
       t.rows_processed  
from v$sql t  
where sql_id in ('fq0w89msgzvs');
```

2. 观察当前数据库的版本及等待情况，SQL 基本情况

--等待事件（当前）

```
select t.event, count(*)  
from v$session t  
group by event  
order by count(*) desc;
```

--等待事件（历史汇集）

```
select t.event, t. total_waits  
from v$system_event t  
order by total_waits desc;
```

--游标使用情况

```
select inst_id, sid, count(*)  
from gv$open_cursor  
group by inst_id, sid  
having count(*) >= 1000  
order by count(*) desc;
```

--PGA 占用最多的进程

```
select p.spid,  
       p.pid,  
       s.sid,  
       s.serial#,  
       s.status,  
       p.pga_alloc_mem,  
       s.username,  
       s.osuser,
```

```

        s.program
    from v$process p, v$session s
    where s.paddr(+) = p.addr
    order by p.pga_alloc_mem desc;

```

--登录时间最长的 SESSION（同时获取到 spid，方便在主机层面 ps -ef |grep spid 来查看）

```

select *
  from (select t.sid,
               t2.spid,
               t.PROGRAM,
               t.status,
               t.sql_id,
               t.PREV_SQL_ID,
               t.event,
               t.LOGON_TIME,
               trunc(sysdate - logon_time)
        from v$session t, v$process t2
        where t.paddr = t2.ADDR
              and t.type <> 'BACKGROUND'
        order by logon_time)
where rownum <= 20;

```

--物理读和逻辑较多的 SQL

--逻辑读最多

```

select *
  from (select sql_id,
               sql_text,
               s.executions,
               s.last_load_time,
               s.first_load_time,
               s.disk_reads,
               s.buffer_gets
        from v$sql s
        where s.buffer_gets > 300
        order by buffer_gets desc)
where rownum <= 20;

```

--物理读最多

```

select *
  from (select sql_id,

```

收获，不止 Oracle

```
        sql_text,  
        s.executions,  
        s.last_load_time,  
        s.first_load_time,  
        s.disk_reads,  
        s.buffer_gets,  
        s.parse_calls  
    from v$sql s  
where s.disk_reads > 300  
    order by disk_reads desc)  
where rownum<=20;
```

--执行次数最多

```
select *  
    from (select sql_id,  
                sql_text,  
                s.executions,  
                s.last_load_time,  
                s.first_load_time,  
                s.disk_reads,  
                s.buffer_gets,  
                s.parse_calls  
            from v$sql s  
            order by s.executions desc)  
where rownum <= 20;
```

--解析次数最多

```
select *  
    from (select sql_id,  
                sql_text,  
                s.executions,  
                s.last_load_time,  
                s.first_load_time,  
                s.disk_reads,  
                s.buffer_gets,  
                s.parse_calls  
            from v$sql s  
            order by s.parse_calls desc)  
where rownum <= 20;
```

--求 DISK SORT 严重的 SQL

```
select sess.username, sql.sql_text, sql.address, sort1.blocks  
    from v$session sess, v$sqlarea sql, v$sort_usage sort1
```

```

where sess.serial# = sort1.session_num
  and sort1.sqladdr = sql.address
  and sort1.sqlhash = sql.hash_value
  and sort1.blocks > 200
order by sort1.blocks desc;

```

3. 检查是否有过分提交的语句

检查是否有过分提交的语句，关键是得到 sid，代入 V\$SESSION 就可知道是什么进程，接下来还可以知道 V\$SQL --提交次数最多的 SESSION

```

select t1.sid, t1.value, t2.name
  from v$sesstat t1, v$statname t2
--where t2.name like '%commit%'
where t2.name like '%user commits%' --可以只选 user commits,其他系统级的先不关心
  and t1.STATISTIC# = t2.STATISTIC#
  and value >= 10000
order by value desc;

```

--取得 SID 既可以代入到 V\$SESSION 和 V\$SQL 中去分析

--得出 SQL_ID

```

select t.sid,
       t.program,
       t.machine,
       t.logon_time,
       t.wait_class,
       t.wait_time,
       t.seconds_in_wait,
       t.event,
       t.sql_id,
       t.prev_sql_id
  from v$session t
where sid in(1404,1388,2022,2073,2175,1075,1832,1072,2109,1807,1764,1485,1664);

```

--根据 sql_id 或 prev_sql_id 代入得到 SQL

```

select t.sql_id,
       t.sql_text,
       t.executions,
       t.first_load_time,
       t.last_load_time,
       t.buffer_gets,
       t.rows_processed

```

收获，不止 Oracle

```
from v$sql t
where sql_id in ('');
```

4. 检查系统使用绑定变量的情况

--查询共享内存占有率

```
select count(*),round(sum(sharable_mem)/1024/1024,2)
from   v$db_object_cache  a;
```

--捕获出需要使用绑定变量的 SQL（这里只能适配大多数语句）

```
Drop table t1 purge;
```

```
create table t1 as select sql_text,module from v$sqlarea;
alter table t1 add sql_text_wo_constants varchar2(1000);
create or replace function
remove_constants( p_query in varchar2 ) return varchar2
as
    l_query long;
    l_char  varchar2(10);
    l_in_quotes boolean default FALSE;
begin
    for i in 1 .. length( p_query )
    loop
        l_char := substr(p_query,i,1);
        if ( l_char = "'" and l_in_quotes )
        then
            l_in_quotes := FALSE;
        elsif ( l_char = "'" and NOT l_in_quotes )
        then
            l_in_quotes := TRUE;
            l_query := l_query || "'#";
        end if;
        if ( NOT l_in_quotes ) then
            l_query := l_query || l_char;
        end if;
    end loop;
    l_query := translate( l_query, '0123456789', '@@@@@@@@@@' );
    for i in 0 .. 8 loop
        l_query := replace( l_query, lpad('@',10-i,'@'), '@' );
        l_query := replace( l_query, lpad(' ',10-i,' '), ' ');
    end loop;
    return upper(l_query);
```

```

end;
/
update t1 set sql_text_wo_constants = remove_constants(sql_text);
commit;

---执行完上述动作后，以下 SQL 语句可以完成未绑定变量语句的统计
select sql_text_wo_constants, module, count(*)
  from t1
 group by sql_text_wo_constants, module
 having count(*) > 100
 order by 3 desc;

```

11.2.4.3 静态整体

静态整体问题的解决流程如图 11-9 所示。

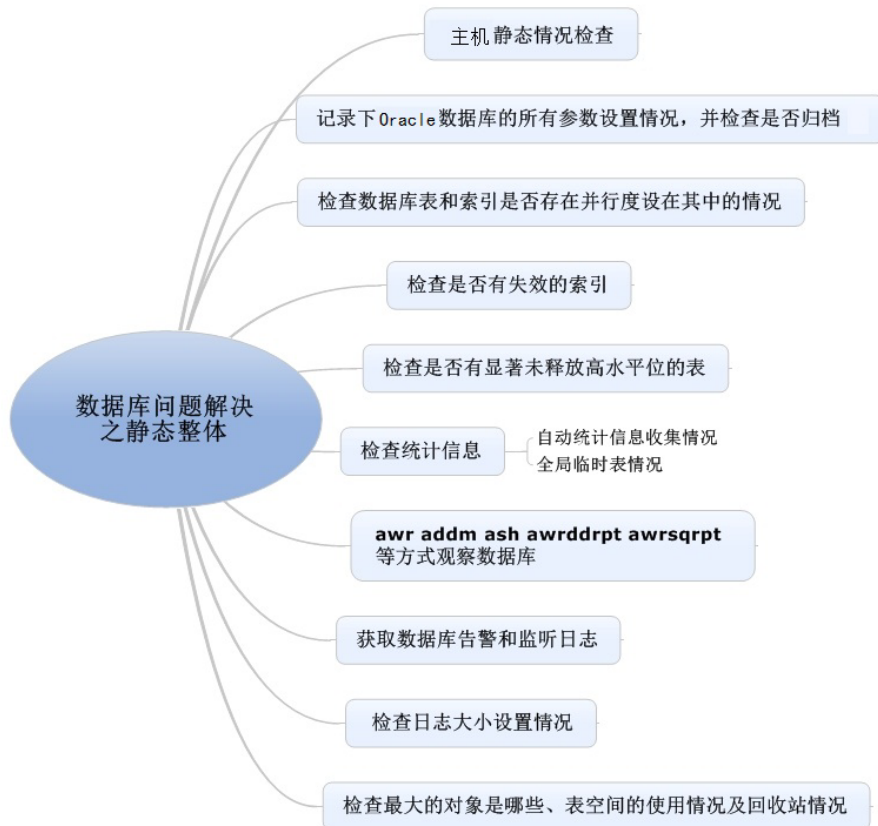


图 11-9 数据库问题解决之静态整体

收获，不止 Oracle

1. 主机静态情况检查

查看磁盘空间使用情况

```
(AIX)    df -g
(Linux)   df -h
(HP-UNIX) bdf
```

主机内存情况

```
(AIX)    lsattr -El sys0 -a realmem
(Linux)   cat /proc/meminfo
(HP-UNIX) machinfo
```

主机 CPU 情况

```
(AIX)    lsdev -C |grep proc
(Linux)   cat /proc/cpuinfo
(HP-UNIX) machinfo
(所有都支持) vmstat 1 6
```

2. 记录下 Oracle 数据库的所有参数设置情况，并检查是否归档

--版本及所有参数情况

开启 CRT 的日志跟踪

sqlplus "/ as sysdba"

--版本

select * from v\$version;

--所有参数

show parameter

关闭 CRT 的日志跟踪，将文件取回

其中重点关注的是

sga pga log_buffer

processes open_cursors session_cached_cursors

db_recovery_file_dest

cluster_database

--是否归档

archive log list

3. 检查数据库表和索引是否存在并行度设在其中的情况

检查数据库表和索引是否存在并行度设在其中的情况(很多时候有人用 parallel 建了表或索引,忘记 alter table xxx noparallel 关闭了)。

```
select t.owner, t.table_name, degree
  from dba_tables t
 where t.degree > '1';
```

```
select t.owner, t.table_name, index_name, degree, status
```

```

from dba_indexes t
where owner in ('LJB')
and t.degree > '1';

--有问题就要处理，比如索引有并行，就处理如下：
select 'alter index ' || t.owner || '.' || index_name || ' noparallel;'
from dba_indexes t
where owner in ('LJB')
and t.degree > '1';

```

4. 检查是否有失效的索引

--普通索引

```

select t.index_name,
       t.table_name,
       t.blevel,
       t.num_rows,
       t.leaf_blocks,
       t.distinct_keys
from dba_indexes t
where status = 'INVALID';

```

--分区索引

```

select  t2.owner,
        t1.blevel,
        t1.leaf_blocks,
        t1.index_name,
        t2.table_name,
        t1.partition_name,
        t1.status
from dba_ind_partitions t1, dba_indexes t2
where t1.index_name = t2.index_name
and t1.status = 'UNUSABLE'
and t2.owner in ('LJB');

```

---以下是所有失效对象的检查

```

select 'alter ' ||
       decode(object_type, 'PACKAGE BODY', 'PACKAGE', 'TYPE BODY', 'TYPE', object_type) || ' ' ||
       owner || '.' || object_name || ' ' ||
       decode(object_type, 'PACKAGE BODY', 'compile body', 'compile') || ';',
t.*
from dba_objects t
where status='INVALID'

```

收获，不止 Oracle

```
--owner not in ('PUBLIC', 'SYSTEM', 'SYS')
and owner in ('LIB');
```

5. 检查是否有显著未释放高水平位的表

```
select table_name,blocks,num_rows
  from user_tables
 where blocks / num_rows >= 0.2
    and num_rows is not null
    and num_rows <>0
    and blocks>=10000;
```

这个就可以预测到哪些是高水平位没释放的表。

其中 blocks>=10000 是因为低于 10000 的块说明表的体积太小了，释放或不释放无所谓。

而 blocks / num_rows >= 1 表示是严重有问题的。

而这个 blocks / num_rows >= 0.2 表示至少一个块要装 5 行数据，如果装不了，那就很奇怪了，值得怀疑了，除非有 LONG 和 CLOB 字段或者一堆的 VARCHAR2(4000)字段。

附（可以释放高水平位的脚本，在 Oracle 的 shrink 方法无效时可采纳）：

```
create or replace package pkg_shrink
Authid Current_User
as
/*
created by ljb at 2010-10-18
功能：将 delete 后的表降低高水平
*/
procedure p_move_tab (p_tab varchar2);
procedure p_cal_bytes (p_status varchar2 ,p_tab varchar2) ;
procedure p_rebuild_idx(p_tab varchar2);
procedure p_main(p_table_name varchar2);
end   pkg_shrink ;
/
create or replace package body pkg_shrink
as

v_sql   varchar2(4000);

procedure   p_cal_bytes (p_status varchar2 ,p_tab varchar2)
as
v_tab_bytes number;
```

```

v_idx_bytes number;
v_str_tab      varchar2(4000);
v_str_idx      varchar2(4000);
begin
    select sum(bytes)/1024/1024 into v_tab_bytes from user_segments where segment_name=upper(p_tab);
    select sum(bytes)/1024/1024 into v_idx_bytes from user_segments where segment_name IN (SELECT
INDEX_NAME FROM USER_INDEXES WHERE TABLE_NAME=upper(p_tab));
    v_str_tab:=p_status||'表'||p_tab||'的大小为'||v_tab_bytes||'M';
    if v_idx_bytes is null then
        v_str_idx:=p_status||'无索引';
    else
        v_str_idx:=p_status||'索引的大小为'||v_idx_bytes||'M';
    end if;
    dbms_output.put_line(v_str_tab || ';' || v_str_idx);
end p_cal_bytes;

procedure p_move_tab (p_tab varchar2)
as
V_IF_PART_TAB NUMBER;
begin
    SELECT COUNT(*) INTO V_IF_PART_TAB FROM user_part_tables WHERE TABLE_NAME=upper(P_TAB);
    IF V_IF_PART_TAB=0 THEN  ----非分区表
        v_sql:='alter table '||p_tab ||' move';--完成表的 MOVE 动作，从而做到降低高水平位，不过也带来了索引
的失效！
        DBMS_OUTPUT.put_line(v_sql);
        execute immediate v_sql;
    ELSE  ----分区表
        for i in (SELECT * from USER_TAB_PARTITIONS WHERE TABLE_NAME=upper(p_tab)) loop
            v_sql:='alter table '|| p_tab ||' move partition ' ||i.partition_name; --完成分区表的 MOVE 动作，同样带来了索引失效！
            DBMS_OUTPUT.put_line(v_sql);
            execute immediate v_sql;
        end loop;
    END IF;
end p_move_tab;

procedure p_rebuild_idx(p_tab varchar2)
as
V_NORMAL_IDX NUMBER;
V_PART_IDX NUMBER;
begin
    SELECT COUNT(*) INTO V_NORMAL_IDX FROM user_indexes where table_name='PART_TAB' AND INDEX_NAME
NOT IN (SELECT INDEX_NAME FROM user_part_indexes);

```

收获，不止 Oracle

```
IF V_NORMAL_IDX>=1 THEN  ---普通索引
    for i in (select * from user_indexes where table_name=upper(p_tab) AND INDEX_NAME
NOT IN (SELECT INDEX_NAME FROM user_part_indexes)) loop
        v_sql:= 'alter index '||i.index_name ||' rebuild'; --将失效的普通索引重建
        DBMS_OUTPUT.put_line(v_sql);
        execute immediate v_sql;
    end loop;
END IF;
SELECT COUNT(*) INTO V_PART_IDX FROM user_part_indexes WHERE TABLE_NAME='PART_TAB';
IF V_PART_IDX>=1  THEN  ---分区索引
    for i in (SELECT * from User_Ind_Partitions WHERE index_name in (select index_name from
user_part_indexes where table_name =upper(p_tab))) loop
        v_sql:= 'alter index '||i.index_name ||' rebuild partition ' ||i.partition_name; ---将失效分区索引重建
        DBMS_OUTPUT.put_line(v_sql);
        execute immediate v_sql;
    end loop;
END IF;
end p_rebuild_idx;

procedure p_main(p_table_name varchar2)
as
begin
    for i in (select * from (
        SELECT SUBSTR(s,INSTR(s,',',1,ROWNUM)+1, INSTR(s,',',1,ROWNUM+1) - INSTR(s,',',1,ROWNUM)-1) AS
TYPE_ID
        FROM (SELECT ','||p_table_name||',' AS s FROM DUAL)
        CONNECT BY ROWNUM<=100
    )
    WHERE type_id IS NOT NULL
    ) loop  ---在外面 SELECT 再套一层是必须的，否则只会循环一次。另外 type_id IS NOT NULL 是必须的，
否则会多循环
        DBMS_OUTPUT.put_line('当前处理的表为'||I.TYPE_ID);
        p_cal_bytes('未降低高水平位前',i.type_id);
        p_move_tab(i.type_id);
        p_rebuild_idx(I.TYPE_ID);
        p_cal_bytes('降低高水平位后',i.type_id);
    end loop;

end p_main;

end pkg_shrink;
/
```

6. 检查统计信息

(1) 自动统计信息收集情况

检查哪些对象的统计信息不够新，或者从未统计过（注意，让未统计过的在前面，即 `nulls first`）。

--检查统计信息是否被收集

```
select t.JOB_NAME,t.PROGRAM_NAME,t.state,t.enabled
  from dba_scheduler_jobs t
 where job_name = 'GATHER_STATS_JOB';
```

--检查哪些未被收集或者很久没收集

```
select owner,
       table_name,
       t.last_analyzed,
       t.num_rows,
       t.blocks,
       t.object_type
  from dba_tab_statistics t
 where owner in ('LJB')
    and (t.last_analyzed is null or t.last_analyzed <= sysdate - 14)
 order by t.last_analyzed nulls first;
```

--查看数量

```
select count(*)
  from dba_tab_statistics t
 where owner in ('LJB')
    and (t.last_analyzed is null or t.last_analyzed <= sysdate - 14);
```

(2) 全局临时表情况

--检查全局临时表有没有被收集统计信息

```
select owner,
       table_name,
       t.last_analyzed,
       t.num_rows,
       t.blocks
  from dba_tables t
 where t.temporary = 'Y'
    and owner in ('LJB');
```

---相应的处理措施

```
BEGIN
DBMS_STATS.DELETE_TABLE_STATS('LJB','RN_IDENTIFICATION_BATCH');--删除统计信息
```

收获，不止 Oracle

```
DBMS_STATS.LOCK_TABLE_STATS('LJB','RN_IDENTIFICATION_BATCH'); --不收集统计信息
END;
```

7. awr addm ash awrddrpt awrsqrpt 等方式观察数据库

```
exec dbms_workload_repository.create_snapshot();
sqlplus "/ as sysdba"
@?/rdbms/admin/awrrpt.sql
@?/rdbms/admin/addmrpt.sql
@?/rdbms/admin/ashrpt.sql
@?/rdbms/admin/awrddrpt.sql
@?/rdbms/admin/awrsqrpt.sql
```

注意：一般要考虑统计出问题的时间段的报表，顺序一般是 awr→addm→ash→awrdd→awrsq

8. 获取数据库告警和监听日志

lsnrctl status 可获取监听的路径

```
sqlplus "/ as sysdba"
```

show parameter ump 可获取告警日志的路径

文件很大的情况下，可以考虑 tail -n 50000 alert* > alert.log 的方式获取最近 5 万条记录

监听也是类似 tail -n 50000 list* > listener.log

9. 检查日志大小设置情况

一般情况下，网管项目建议单个 REDO 设置为 5GB 大，如果发现告警日志切换频繁，则应该立即调整

```
select inst_id,group#,member from gv$logfile;
```

```
select group#,bytes,status from v$log;
```

10. 检查最大的对象是哪些、表空间的使用情况及回收站情况

--用户的权限情况

```
select * from dba_role_privs where grantee='LJB';
```

--最大的前 20 个对象（然后再进一步 COUNT(*)统计其记录数）

```
select *
  from (select owner,
               segment_name,
               segment_type,
               sum(bytes) / 1024 / 1024 / 1024 object_size
          from DBA_segments
         WHERE owner in('LJB')
        group by owner,segment_name, segment_type)
```

```

    order by object_size desc)
where rownum <= 50;

--表空间使用情况
select a.tablespace_name "表空间名",
       a.total_space "总空间(g)",
       nvl(b.free_space, 0) "剩余空间(g)",
       a.total_space - nvl(b.free_space, 0) "使用空间(g)",
       trunc(nvl(b.free_space, 0) / a.total_space * 100, 2) "剩余百分比%"
from (select tablespace_name,
             trunc(sum(bytes) / 1024 / 1024 / 1024, 2) total_space
      from dba_data_files
      group by tablespace_name) a,
     (select tablespace_name,
             trunc(sum(bytes / 1024 / 1024 / 1024), 2) free_space
      from dba_free_space
      group by tablespace_name) b
where a.tablespace_name = b.tablespace_name(+)
order by 5;

--整个用户有多大（比如 LJB 用户）
select sum(bytes)/1024 /1024 /1024 "G"
   from dba_segments
where owner = 'LJB';

--回收站情况
select SUM(BYTES) / 1024 / 1024 / 1024
   from DBA_SEGMENTS
WHERE owner = 'LJB'
      AND SEGMENT_NAME LIKE 'BIN$%';

--分区最多的前 20 个对象（先知道表就可以大概了解了，索引可以后续再观察）
select *
   from (select table_owner, table_name, count(*) cnt
        from dba_tab_partitions
        WHERE table_owner in ('LJB')
        group by table_owner, table_name
        order by cnt desc)
where rownum <= 20;

```


11.2.4.4 静态局部

静态局部问题的解决流程如图 11-10 所示。

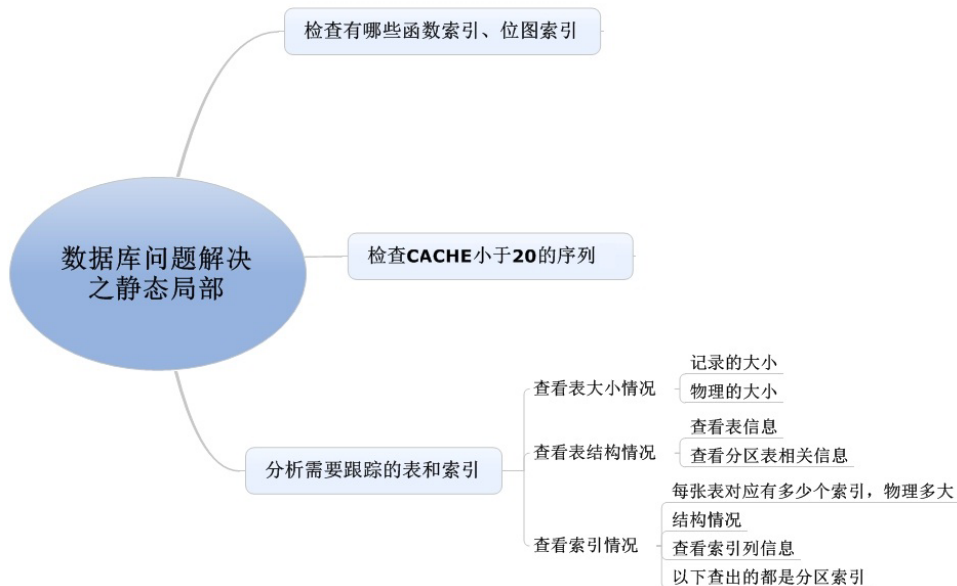


图 11-10 数据库问题解决之静态局部

1. 检查有哪些函数索引或者位图索引

检查有哪些函数索引或者位图索引（大多数情况下开发人员对这两类索引是使用不当的，所以需要捞出来确认一下）

```
select t.owner,
       t.index_name,
       t.index_type,
       t.status,
       t.blevel,
       t.leaf_blocks
from dba_indexes t
where index_type in ('BITMAP', 'FUNCTION-BASED NORMAL')
and owner in ('LJB'); ---比如 LJB 这个用户
```

2. 检查 CACHE 小于 20 的序列

检查序 CACHE 小于 20 的序列的情况（一般情况下可将其增至 1000 左右，序列默认的 20 太小）

```
select t.sequence_name,
       t.cache_size,
       'alter sequence ' || t.sequence_owner || '.' || t.sequence_name ||
       ' cache 1000;'
```

```

from dba_sequences t
where sequence_owner in ('LJB')
AND CACHE_SIZE <= 20;

```

3. 分析需要跟踪的表和索引的情况

(1) 查看表大小情况

记录的大小

```

select count(*) from NOTIFICATION_INTERFACE;
select count(*) from INP_IDEP_DATA_FILES;
select count(*) from NE_ALARM_LIST;

```

物理的大小

```

select segment_name,sum(bytes)/1024/1024
from user_segments
where segment_name in ('NOTIFICATION_INTERFACE',
                        'INP_IDEP_DATA_FILES',
                        'NE_ALARM_LIST')
group by segment_name;

```

(2) 查看表结构情况

--查看表信息

```

select
t.table_name,
  t.num_rows,
  t.blocks,
  -- t.empty_blocks,--统计信息不收集这个字段，所以不需要这个字段了
  t.degree,
  t.last_analyzed,
  t.temporary,
  t.partitioned,
  t.pct_free,
  t.tablespace_name
from user_tables t
where table_name in ('NOTIFICATION_INTERFACE',
                    'INP_IDEP_DATA_FILES',
                    'NE_ALARM_LIST') ;

```

查看分区表相关信息

--查看分区表相关信息（user_part_tables 记录分区的表的信息，user_tab_partitions 记录表的分区的信息）

--以下了解这些表的分区是什么类型的，有多少个分区

```

select t.table_name,
       t.partitioning_type,
       t.partition_count
from user_part_tables t

```

收获，不止 Oracle

```
where table_name in
  ('NOTIFICATION_INTERFACE',
   'INP_IDEP_DATA_FILES',
   'NE_ALARM_LIST');

--以下了解这些表以什么列作为分区
Select name,
       object_type, column_name
from user_part_key_columns
where name in
  ('NOTIFICATION_INTERFACE',
   'INP_IDEP_DATA_FILES',
   'NE_ALARM_LIST');

--以下了解这些表的分区范围是多少
SELECT table_name,partition_name, high_value, tablespace_name
  FROM user_tab_partitions t
 where table_name in
    ('NOTIFICATION_INTERFACE',
     'INP_IDEP_DATA_FILES',
     'NE_ALARM_LIST')
 order by table_name,t.partition_position;
```

(3) 每张表对应有多少个索引，物理多大

```
select t2.table_name,
       t1.segment_name,
       sum(t1.bytes) / 1024 / 1024
  from user_segments t1, user_indexes t2
 where t1.segment_name = t2.index_name
       and t1.segment_type like '%INDEX%'
       and t2.table_name in
         ('NOTIFICATION_INTERFACE', 'INP_IDEP_DATA_FILES', 'NE_ALARM_LIST')
 group by t2.table_name, t1.segment_name
 order by table_name;

--结构情况（高度、重复度、并行度、叶子高度、聚合因子、记录数、状态、最近分析时间……）
select t.table_name,
       t.index_name,
       t.num_rows,
       t.index_type,
       t.status,
       t.clustering_factor,
       t.blevel,
```

```

t.distinct_keys,
t.leaf_blocks,
t.uniqueness,
t.degree,
t.last_analyzed
from user_indexes t
where table_name in ('NOTIFICATION_INTERFACE',
                    'INP_IDEP_DATA_FILES',
                    'NE_ALARM_LIST');

```

(4) 查看索引列信息

---以下可以查出来的是索引的列是什么（无论分区表和非分区表都可以查出）

```

select  t.table_name,t.index_name, t.column_name, t.column_position, t.DESCE
from user_ind_columns t
where table_name in('NOTIFICATION_INTERFACE',
                    'INP_IDEP_DATA_FILES',
                    'NE_ALARM_LIST')
order by table_name,index_name, column_position;

```

--以下查出的都是分区索引

```

select  table_name,index_name, partitioning_type, partition_count
from user_part_indexes
where table_name in ('NOTIFICATION_INTERFACE',
                    'INP_IDEP_DATA_FILES',
                    'NE_ALARM_LIST')
order by table_name,index_name;

```

```

select index_name, partition_name, status,blevel, leaf_blocks
from user_ind_partitions
where index_name in
(select index_name
 from user_indexes
 where table_name in ('NOTIFICATION_INTERFACE',
                    'INP_IDEP_DATA_FILES',
                    'NE_ALARM_LIST'));

```

11.2.5 开发规范——让开发者驾轻就熟

高效开发规范如图 11-11 所示。

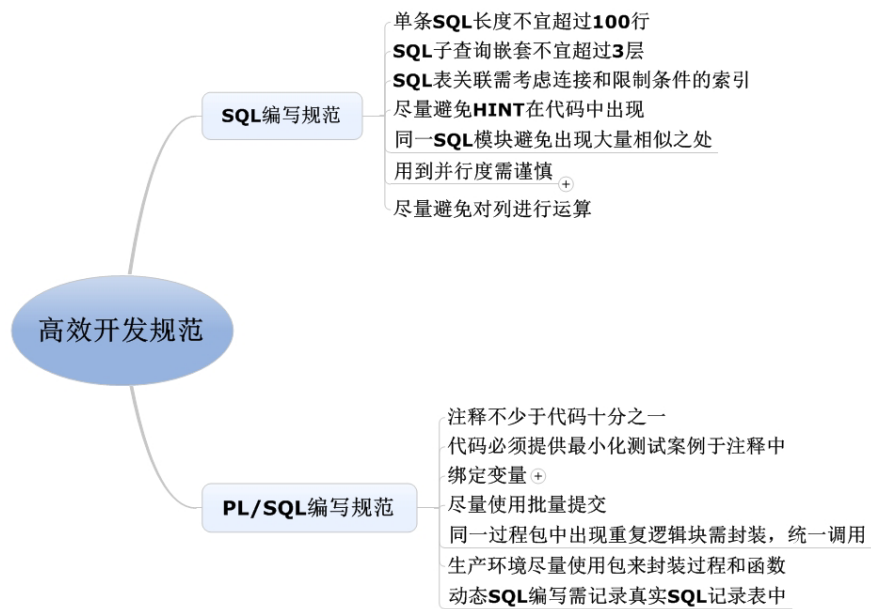


图 11-11 高效开发规范

11.2.5.1 SQL 编写规范

SQL 编写规范如图 11-12 所示。

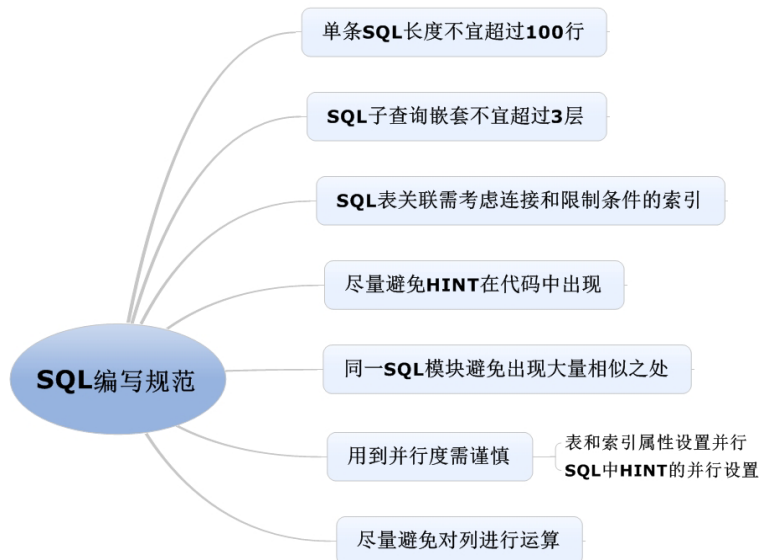


图 11-12 高效开发规范之 SQL 编写规范

1. 单条 SQL 长度不宜超过 100 行

```
select sql_id, count(*)
  from v$sqltext
 group by sql_id
having count(*) >= 100
 order by count(*) desc;
```

脚本 11-1 判断过长的 SQL

2. SQL 子查询嵌套不宜超过 3 层

一般来说，子查询嵌套如果超过 3 层，容易导致 SQL 语句的解析过于复杂，导致产生错误的执行计划。此外子查询嵌套过多，一般适宜分解成更简单的多条 SQL。

3. SQL 表关联需考虑连接和限制条件的索引

```
drop table t purge;
create table t as select * from v$sql_plan;

select *
  from t
 where sql_id not in (select sql_id
                      from t
                      where sql_id in (select sql_id from t where operation = 'NESTED LOOPS' )
                      and (operation like '%INDEX%' or object_owner like '%SYS%'))
 and sql_id in
  (select sql_id from t where sql_id in (select sql_id from t where operation = 'NESTED LOOPS'));
```

脚本 11-2 使用 Nested Loops Join 但是未用到索引的，比较可疑

4. 尽量避免 HINT 在代码中出现

```
select sql_text,
       sql_id,
       module,
       t.service,
       first_load_time,
       last_load_time,
       executions
  from v$sql t
 where sql_text like '%/*+%'
 and t.SERVICE not like 'SYS$%';
```

脚本 11-3 找出非 SYS 用户用 HINT 的所有 SQL 来分析

收获，不止 Oracle

5. 同一 SQL 模块避免出现大量相似之处

这种 SQL 写法一般比较可疑，一般可以优化，比如 WITH 子句等等，所以出现后需引起注意。

6. 用到并行度需谨慎

(1) 表和索引属性设置并行

```
select t.owner, t.table_name, degree
  from dba_tables t
 where t.degree > '1';

select t.owner, t.table_name, index_name, degree, status
  from dba_indexes t
 where owner in ('LJB')
    and t.degree > '1';
```

--有问题就要处理，比如索引有并行，就处理如下：

```
select 'alter index ' || t.owner || '.' || index_name || ' noparallel;'
  from dba_indexes t
 where owner in ('LJB')
    and t.degree > '1';
```

脚本 11-4 找出被设置成并行属性的表和索引，并修正

(2) SQL 中 HINT 的并行设置

```
select sql_text,
       sql_id,
       module,
       .service,
       first_load_time,
       last_load_time,
       executions
  from v$sql t
 where sql_text like '%parall%'
    and t.SERVICE not like 'SYS$%';
```

脚本 11-5 属性未设并行，但是 HINT 设并行的 SQL

7. 尽量避免对列进行运算

```
select sql_text,
       sql_id,
       module,
       t.service,
       first_load_time,
```

```
last_load_time,
executions
from v$sql t
where (upper(sql_text) like '%TRUNC%'
      or upper(sql_text) like '%TO_DATE%'
      or upper(sql_text) like '%SUBSTR%')
and t.SERVICE not like 'SYS$%';
```

脚本 11-6 捞取对列进行运算的 SQL

11.2.5.2 PL/SQL 编写规范

PL/SQL 编写规范如图 11-13 所示。

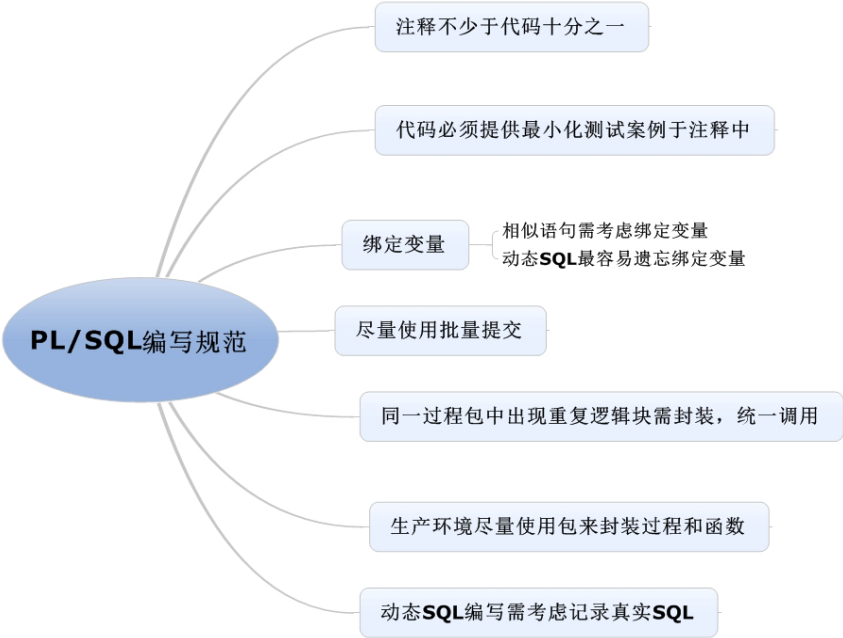


图 11-13 高效开发规范之 PL/SQL 编写规范

1. 注释不少于代码的十分之一

注释如果过少，将容易导致开发者后续忘记代码的逻辑，尤其是对新人交接很不利，一般建议注释不少于代码的十分之一。

```
select * from (
select name,
t.type,
```



```
sum(case when text like '%--%' then 1 else 0 end) / count(*) rate
from user_source t
where type in ('package body', 'procedure', 'function')---包头就算了
group by name, type
having sum(case when text like '%--%' then 1 else 0 end) / count(*)<=1/10)
order by rate;
```

脚本 11-7 捞取注释少于代码十分之一的程序

2. 代码必须提供最小化测试案例于注释中

这是一个值得推崇的好习惯，对新人接手熟悉程序尤为有用。

3. 绑定变量

(1) 相似语句需考虑绑定变量

这里就不提供脚本了，在 11.2.4.2 节的“检查系统使用绑定变量的情况”中已提供代码。

(2) 动态 SQL 最容易遗忘绑定变量

一般来说，动态 SQL 未用绑定变量的情况多半是因为未使用 USING 关键字，所以可用如下脚本来搜索可疑的未用绑定变量的动态 SQL。

```
select *
from user_source
where name in
(select name from user_source where name in (select name from user_source where UPPER(text) like
'%EXECUTE IMMEDIATE%'))
and name in
(select name from user_source where name in (select name from user_source where UPPER(text) like
'%||%''))
and name not in
(select name from user_source where name in (select name from user_source where upper(text) not like
'%USING%'));
```

脚本 11-8 动态 SQL 未用 USING 有可能未用绑定变量

4. 尽量使用批量提交

未使用批量提交，一般都是因为将 commit 写到了循环内。以下语句可查出单个 session 提交次数超过 10000 次的情况，这么多次很可疑，应该捞取出来进行分析。

```
select t1.sid, t1.value, t2.name
from v$sesstat t1, v$statname t2
--where t2.name like '%commit%'
where t2.name like '%user commits%' --可以只选 user commits，其他系统级的先不关心
and t1.STATISTIC# = t2.STATISTIC#
```

```
and value >= 10000
order by value desc;
```

脚本 11-9 查询提交次数过多的 SESSION

- 5. 同一过程包中出现重复逻辑块需封装，统一调用
这是封装的概念，否则修改统一逻辑，代码可能需要修改多处，不利于维护。
- 6. 生产环境尽量使用包来封装过程和函数
一般来说，只要是正式的产品，就必须使用包。

```
select distinct name, type
from user_source
where type in ('PROCEDURE', 'FUNCTION')
order by type;
```

脚本 11-10 查询未用包的程序逻辑

- 7. 动态 SQL 编写需设法保存真实 SQL

动态 SQL 编写最大的麻烦在于调试困难，因为语句是拼装组合而成的，无论是出现该语句的语法错误还是性能问题，都无法被有效跟踪到。此时考虑将拼装成的 SQL 记录在某张表里，将会给调试跟踪带来极大的方便。

11.2.6 设计规范——助设计者运筹帷幄

数据库设计规范如图 11-14 所示。

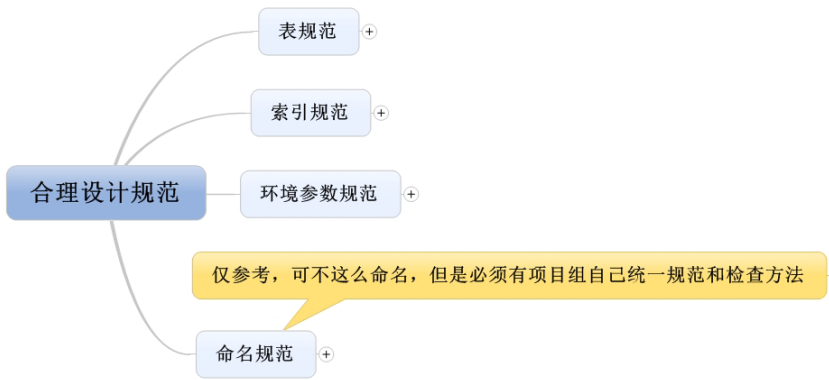


图 11-14 设计规范

11.2.6.1 表规范

设计规范中的表规范如图 11-15 所示。

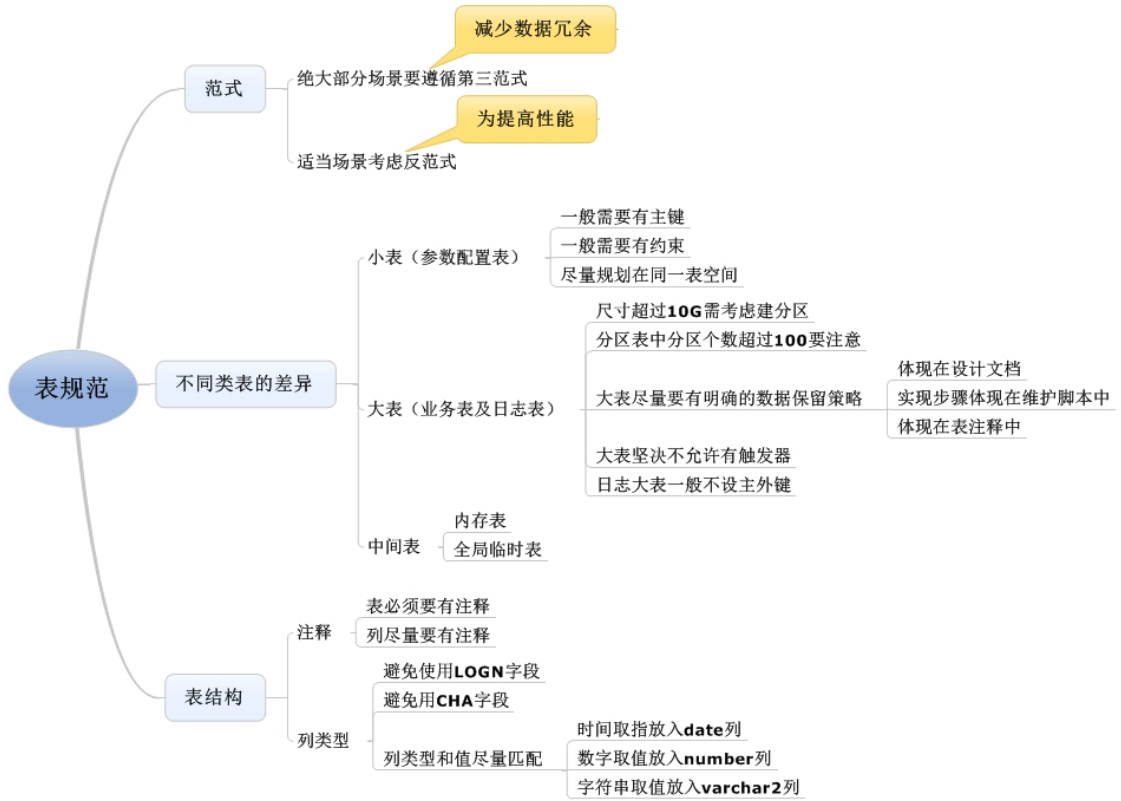


图 11-15 设计规范之表规范

1. 范式

- ① 绝大部分场景要遵循第三范式。

② 适当场景考虑反范式。

2. 不同类表的要点差异

(1) 小表 (参数配置表)

- ① 一般需要有主键。

② 一般需要有约束。

③ 尽量规划在同一表空间。

(2) 大表（业务表及日志表）

1) 尺寸超过**10GB**需考虑建分区

--表大小超过 10GB 未建分区的

```
select owner,
       segment_name,
       segment_type,
       sum(bytes) / 1024 / 1024 / 1024 object_size
from dba_segments
WHERE segment_type = 'TABLE' ---此处说明是普通表，不是分区表，如果是分区表，类型是 TABLE PARTITION
group by owner, segment_name, segment_type
having sum(bytes) / 1024 / 1024 / 1024 >= 10
order by object_size desc;
```

脚本 11-11 大小超过 10GB 未建分区的表

2) 分区表中分区个数超过**100**要注意

--分区个数超过 100 个的表

```
select table_owner, table_name, count(*) cnt
from dba_tab_partitions
WHERE table_owner in ('LJB')
having count(*)>=100
group by table_owner, table_name
order by cnt desc;
```

脚本 11-12 查询分区个数超过 100 的表

3) 大表尽量要有明确的数据保留策略

- ① 体现在设计文档。
- ② 实现步骤体现在维护脚本中。
- ③ 体现在表注释中。

---超过 10GB 的大表没有时间字段

```
select T1.*, t2.column_name, t2.data_type
from (select segment_name,
            segment_type,
            sum(bytes) / 1024 / 1024 / 1024 object_size
from user_segments
WHERE segment_type = 'TABLE' ---此处说明是普通表，不是分区表，如果是分区表，类型是 TABLE
PARTITION
group by segment_name, segment_type
```

收获，不止 Oracle

```
having sum(bytes) / 1024 / 1024 / 1024 >= 0.01
order by object_size desc) t1,
user_tab_columns t2
where t1.segment_name = t2.table_name(+)
and t2.DATA_TYPE = 'DATE' --来说明这个大表有时间列

---上述语句和下面的语句进行观察比较
select segment_name,
       segment_type,
       sum(bytes) / 1024 / 1024 / 1024 object_size
from user_segments
WHERE segment_type = 'TABLE';    ---此处说明是普通表，不是分区表，如果是分区表，类型是 TABLE
PARTITION
group by segment_name, segment_type
having sum(bytes) / 1024 / 1024 / 1024 >= 0.01
order by object_size desc;
```

脚本 11-13 表大小超过 10GB，有时间字段，可考虑在该列建分区

4) 大表坚决不允许有触发器

```
select trigger_name, table_name, tab_size
from user_triggers t1,
(select segment_name, sum(bytes / 1024 / 1024 / 1024) tab_size
 from user_segments t
 where t.segment_type='TABLE'
 group by segment_name) t2
where t1.TABLE_NAME=t2.segment_name;
```

脚本 11-14 找出有建触发器的表，同时观察该表多大

3. 表结构

(1) 注释

1) 表必须要有注释

```
col COMMENTS for a40;
select TABLE_NAME,T.TABLE_TYPE
from USER_TAB_COMMENTS T
where table_name not like 'BIN$%'
and comments is null
order by table_name;
```

脚本 11-15 查询那些表未做注释

2) 列尽量要有注释

```
select TABLE_NAME,COLUMN_NAME
  from USER_COL_COMMENTS
 where table_name not like 'BIN$%'
    and comments isnull
 order by table_name;
```

脚本 11-16 查询哪些列未做注释（仅供参考）

(2) 列类型

1) 避免使用LONG字段

可以说，现在的应用中，LONG字段几乎是有百害而无一利，所以尽量要杜绝在设计中出现LONG，一般考虑CLOB字段来替代。

```
SELECT TABLE_NAME, COLUMN_NAME, DATA_TYPE
  FROM user_tab_columns
 WHERE DATA_TYPE = 'LONG'
 ORDER BY 1, 2;
```

脚本 11-17 查询哪些列是 LONG 类型

2) 避免用CHAR字段

CHAR字段的应用场合非常少，一般现在都考虑用VARCHAR2来替代。

```
SELECT TABLE_NAME, COLUMN_NAME, DATA_TYPE
  FROM user_tab_columns
 WHERE DATA_TYPE = 'CHAR'
 ORDER BY 1, 2
```

脚本 11-18 查询哪些列是 CHAR 类型

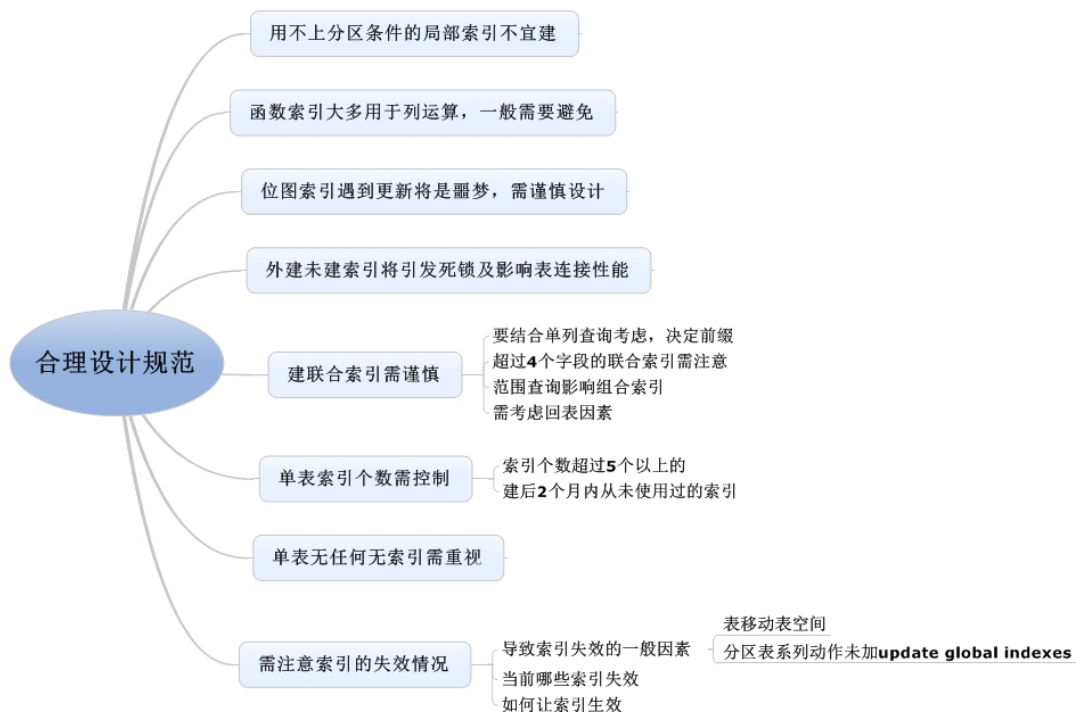
3) 列类型和值尽量匹配

简单三原则：

- ① 时间取值放入 **date** 列。
- ② 数字取值放入 **number** 列。
- ③ 字符串取值放入 **varchar2** 列。

11.2.6.2 索引规范

设计规范中的索引规范如图 11-16 所示。



1. 用不上分区条件的局部索引不宜建

分区表建了分区索引后，如果在查询应用中无法用到这个分区索引列的条件，索引读将可能遍历所有的分区，如果有 100 个分区，相当于遍历了 100 个小索引，将会严重影响性能，此时需要慎重考虑，判断是否需要修改为全局索引。

2. 函数索引大多用于列运算，一般需要避免

从实际应用情况分析，应用函数索引大多是因为设计阶段考虑步骤，比如 `trunc(时间列)` 的写法，往往可以轻易转换成去掉 `trunc` 的写法，所以需要捞取出来验证，如下：

```
select
  t.index_name,
  t.index_type,
  t.status,
  t.blevel,
  t.leaf_blocks
from user_indexes t
where index_type in ('FUNCTION-BASED NORMAL');
```

脚本 11-19 查询哪些索引是函数索引

3. 位图索引遇到更新将是噩梦，需谨慎设计

- ① 位图索引不适合用在表频繁更新的场合。
- ② 位图索引不适合在所在列重复度很低的场合。

因为位图索引的应用比较特殊，适用场合比较少，因此有必要捞取出系统中的位图索引，进行核对检测，如下：

```
select
  t.index_name,
  t.index_type,
  t.status,
  t.blevel,
  t.leaf_blocks
from user_indexes t
where index_type in ('BITMAP');
```

脚本 11-20 查询哪些索引是位图索引

4. 外键未建索引将引发死锁及影响表连接性能

外键未建索引，将有可能导致两个严重的问题：一是更新相关的表产生死锁；二是两表关联查询时性能低下，因此设计中需要谨慎考虑。以下为外键未建索引的表的捞取语句，方便我们分析和确认：

```
select table_name,
  constraint_name,
  cname1 || nvl2(cname2, ',' || cname2, null) ||
  nvl2(cname3, ',' || cname3, null) ||
  nvl2(cname4, ',' || cname4, null) ||
  nvl2(cname5, ',' || cname5, null) ||
  nvl2(cname6, ',' || cname6, null) ||
  nvl2(cname7, ',' || cname7, null) ||
  nvl2(cname8, ',' || cname8, null) columns
from (select b.table_name,
  b.constraint_name,
  max(decode(position, 1, column_name, null)) cname1,
  max(decode(position, 2, column_name, null)) cname2,
  max(decode(position, 3, column_name, null)) cname3,
  max(decode(position, 4, column_name, null)) cname4,
  max(decode(position, 5, column_name, null)) cname5,
  max(decode(position, 6, column_name, null)) cname6,
  max(decode(position, 7, column_name, null)) cname7,
  max(decode(position, 8, column_name, null)) cname8,
```



```
count(*) col_cnt
from (select substr(table_name, 1, 30) table_name,
      substr(constraint_name, 1, 30) constraint_name,
      substr(column_name, 1, 30) column_name,
      position
      from user_cons_columns) a,
      user_constraints b
where a.constraint_name = b.constraint_name
      and b.constraint_type = 'R'
      group by b.table_name, b.constraint_name) cons
where col_cnt > ALL
(select count(*)
 from user_ind_columns i
 where i.table_name = cons.table_name
       and i.column_name in (cname1, cname2, cname3, cname4, cname5,
                             cname6, cname7, cname8)
       and i.column_position <= cons.col_cnt
      group by i.index_name);
```

脚本 11-21 查询外键未建索引的表有哪些

5. 建联合索引需谨慎

(1) 要结合单列查询考虑，决定前缀

如：既可以建立 col1,col2 的联合索引，又可以建立 col2,col1 的联合索引，此时如果存在 col1 列单独查询较多的情况下，一般倾向于建立 col1,col2 的联合索引。

(2) 范围查询影响组合索引

组合查询中，如果有等值条件和范围条件组合的情况，等值条件在前，性能更高。

如：where col1=2 and col2>=100 and col2<=120，此时是 col1,col2 的组合索引性能高过 col2,col1 的组合索引，可以在系统捞出 SQL 进行简单分析，如下：

```
select sql_text,
       sql_id,
       service,
       module,
       t.first_load_time
       t.last_load_time
from v$sql t
where (sql_text like '%>%' or sql_text like '%<%' or sql_text like '%<>%')
      and sql_text not like '%=>%'
      and service not like 'SYS$%';
```

脚本 11-22 将有不等值查询的 SQL 捞取出来分析

（3）需考虑回表因素

一般情况下，如果建索引可以避免回表（在索引中即可完成检测），也可考虑对多列建组合索引，不过组合索引列不宜超过4个。

（4）超过4个字段的联合索引需注意

```
select table_name, index_name, count(*)
  from user_ind_columns
 group by table_name, index_name
 having count(*) >= 4
 order by count(*) desc;
```

脚本 11-23 捞取超过 4 个字段组合的联合索引

6. 单表索引个数需控制

（1）索引个数超过5个以上的

超过5个以上的索引，在表的记录很大时，将会极大地影响该表的更新，因此在表中建索引时需要谨慎考虑，以下是捞取索引个数超过5个表的脚本，如下：

```
select table_name, count(*)
  from user_indexes
 group by table_name
 having count(*) >= 5
 order by count(*) desc;
```

脚本 11-24 单表的索引个数超过 5 个需注意

（2）建后2个月内从未使用过的索引

一般来说，在2个月内从未被用到的索引是多余的索引，可以考虑删除，具体跟踪和定位的方法如下：

```
select 'alter index ' || index_name || ' monitoring usage;'
from user_indexes;
```

然后观察：

```
set linesize 166
col INDEX_NAME for a10
col TABLE_NAME for a10
col START_MONITORING for a25
col END_MONITORING for a25
select * from v$object_usage;
```

--停止对索引的监控，观察 v\$object_usage 状态变化（以某索引 IDX_OBJECT_ID 为例）

```
alter index IDX_OBJECT_ID nomonitoring usage;
```

脚本 11-25 跟踪索引的使用情况，控制索引的数量

7. 单表无任何索引需重视

单表无任何索引的情况一般比较少见，可以捞取出来，再结合 SQL 应用进行分析，观察该表的大小以及是否有时间字段及编码字段这样的适宜建索引的列，分析可以从以下脚本开始：

```
select table_name
  from user_tables
 where table_name not in (select table_name from user_indexes);
```

脚本 11-26 查询无任何索引的表

8. 需注意索引的失效情况

- ① 对表进行 move 操作，会导致索引失效，操作需考虑索引的重建。
- ② 对分区表进行系列操作，如 split、drop、truncate 分区时，容易导致分区表的全局索引失效，需要考虑增加 update global indexes 的关键字进行操作，或者重建索引。
- ③ 分区表 SPLIT 的时候，如果 MAX 区中已经有记录了，这个时候 SPLIT 就会导致有记录的新增分区的局部索引失效。

（1）普通表及分区表的全局索引失效

```
select index_name, table_name, tablespace_name, index_type
  from user_indexes
 where status = 'UNUSABLE';
```

脚本 11-27 查询失效的普通索引

（2）分区表局部索引失效

```
select  t1.index_name,
        t1.partition_name,
        t1.global_stats,
        t2.table_name,
        t2.table_type
  from user_ind_partitions t1, user_indexes t2
 where t2.index_name = t1.index_name
        and t1.status = 'UNUSABLE';
```

脚本 11-28 查询失效的分区局部索引

11.2.6.3 环境参数规范

设计规范中的环境参数规范如图 11-17 所示。

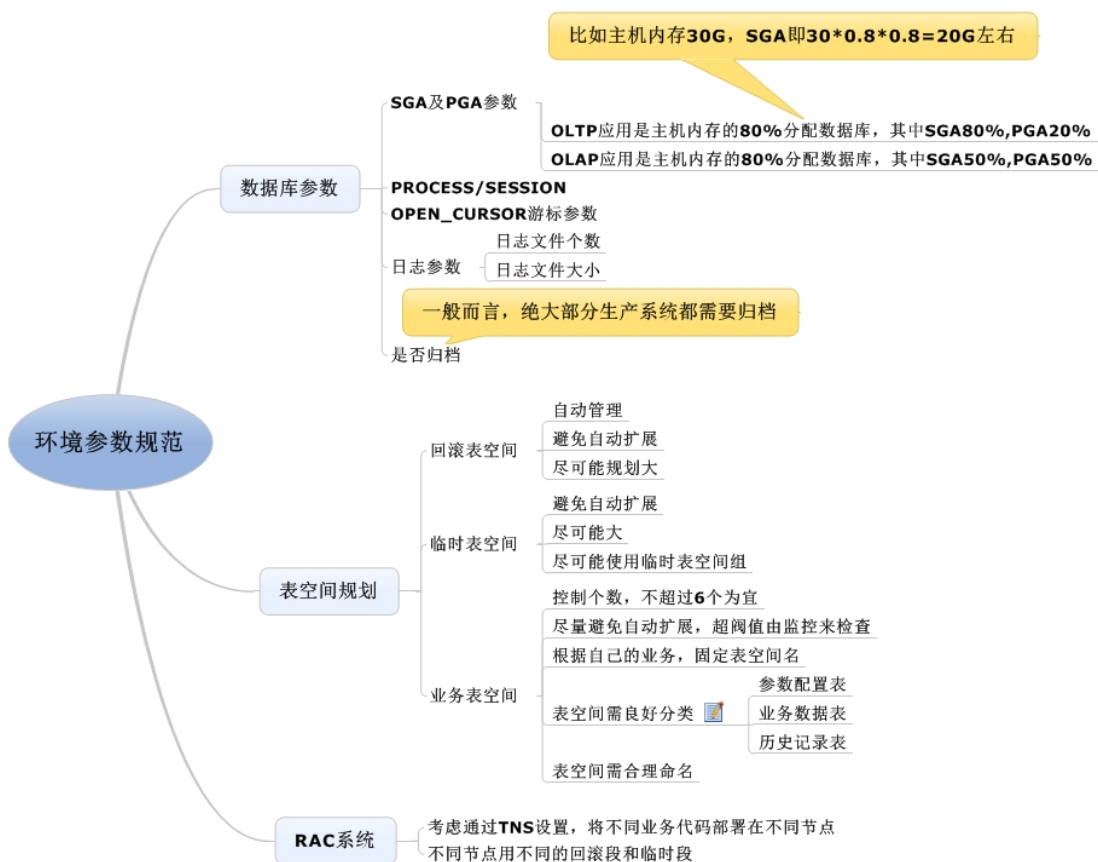


图 11-17 设计规范之环境参数规范

1. 数据库参数

(1) SGA 及 PGA 参数

OLTP 应用是主机内存的 80%分配数据库，其中 SGA80%，PGA20%。

OLAP 应用是主机内存的 80%分配数据库，其中 SGA50%，PGA50%。

如 OLTP 应用：主机内存 30GB，SGA 即是 $30 \times 0.8 \times 0.8 = 20\text{GB}$ 左右。

不过这里还是要注意：并没有什么黄金参数，这些还只能是参考。

(2) PROCESS/SESSION

```
sqlplus "/ as sysdba"
```

收获，不止 Oracle

```
show parameter process
show parameter session
```

```
select count(*) from v$process;
select count(*) from v$session;
```

----默认连接数是 150，这对大多数应用都无法满足，大型应用一般不少于 1000 个。

(3) OPEN_CURSOR 游标参数

```
sqlplus "/ as sysdba"
show parameter open_cursor
```

----默认 open_cursors 是 300，大型应用需设置 1000 以上，原则上不超过 PROCESS 设置。

(4) 日志参数

一般来说，Oracle 默认的日志参数是 3 组，大小为 500MB，在实际较大的生产应用中往往不够，需要至少考虑在 5 组以上，大小在 1GB 以上。

```
archive log list
```

--生产系统大多需要开启归档。

2. 表空间规划

(1) 回滚表空间

- ① 自动管理。
- ② 避免自动扩展。
- ③ 尽可能规划大一些。

(2) 临时表空间

- ① 避免自动扩展。
- ② 尽可能大。
- ③ 尽可能使用临时表空间组。

(3) 业务表空间

- ① 控制个数，不超过 6 个为宜。
- ② 尽量避免自动扩展，超阈值由监控来检查。
- ③ 根据自己的业务，固定表空间名。
- ④ 表空间需良好分类（参数配置表，业务数据表，历史记录表）。
- ⑤ 表空间需合理命名。

3. RAC 系统

- ① 尽量采用 BALANCE 模式，保证两节点压力大致相当。
- ② 可适当考虑不同类型的业务部署在不同的节点上，避免 RAC 的 CACHE 争用。
- ③ 尽量考虑不同的节点使用不同的临时表空间。

11.2.6.4 命名规范

设计规范中的命名规范如图 11-18 所示。

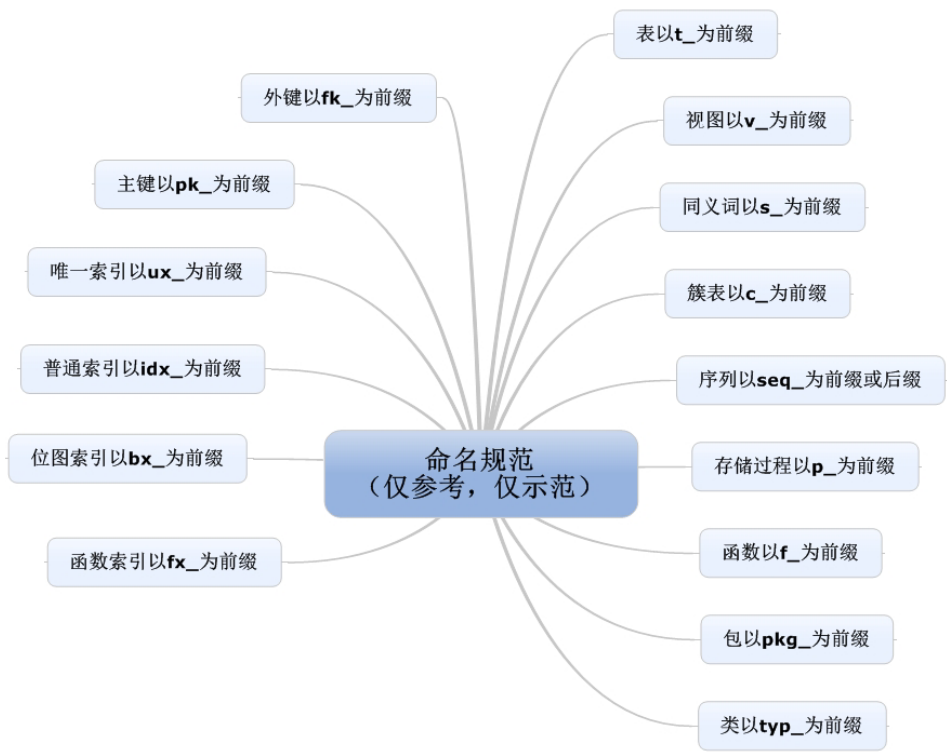


图 11-18 设计规范之命名规范

特别说明，这只是一个示范，大家在现实应用中不一定非要依照下述命名规范，可以有自己的命名规范。但是我强调的是，在同一部门甚至同一公司，要有统一的规范，并且有可以核查的脚本，以下思路仅供参考。

1. 表以 t_ 为前缀

```
select * from user_tables where substr(table_name,1,2)<>'T_';
```

脚本 11-29 查询表的前缀是否以 t 开头

收获，不止 Oracle

2. 视图以 v_为前缀

```
select view_name from user_views where substr(view_name,1,2)<>'V_';
```

脚本 11-30 查询视图的前缀是否以 v 开头

3. 同义词以 s_为前缀

```
select synonym_name, table_owner, table_name  
from user_synonyms  
where substr(synonym_name, 1, 2) <> 'S_';
```

脚本 11-31 查询同义词的前缀是否以 s 开头

4. 簇表以 c_为前缀

```
select t.cluster_name,t.cluster_type  
from user_clusters t  
where substr(cluster_name, 1, 2) <> 'C_';
```

脚本 11-32 查询簇表的前缀是否以 c 开头

5. 序列以 seq_为前缀或后缀

```
select sequence_name,cache_size  
from user_sequences  
where sequence_name not like '%SEQ%';
```

脚本 11-33 查询序列的前缀是否以 seq 开头或结尾

6. 存储过程以 p_为前缀

```
select object_name,procedure_name  
from user_procedures  
where object_type = 'PROCEDURE'  
and substr(object_name, 1, 2) <> 'P_';
```

脚本 11-34 查询存储过程是否以 p 开头

7. 函数以 f_为前缀

```
select object_name,procedure_name  
from user_procedures  
where object_type = 'FUNCTION'  
and substr(object_name, 1, 2) <> 'F_';
```

脚本 11-35 查询函数是否以 f 开头

8. 包以 pkg_ 为前缀

```
select object_name,procedure_name
from user_procedures
where object_type = 'PACKAGE'
and substr(object_name, 1, 4) <> 'PKG_';
```

脚本 11-36 查询包是否以 pkg 开头

9. 类以 typ_ 为前缀

```
select object_name,procedure_name
from user_procedures
where object_type = 'TYPE'
and substr(object_name, 1, 4) <> 'TYP_';
```

脚本 11-37 查询类是否以 typ 开头

10. 主键以 pk_ 为前缀

```
select constraint_name, table_name
from user_constraints
where constraint_type = 'P'
and substr(constraint_name, 1, 3) <> 'PK_'
and constraint_name not like 'BIN$%';
```

脚本 11-38 查询主键是否以 pk 开头

11. 外键以 fk_ 为前缀

```
select constraint_name,table_name
from user_constraints
where constraint_type = 'R'
and substr(constraint_name, 1, 3) <> 'FK_'
and constraint_name not like 'BIN$%';
```

脚本 11-39 查询外键是否以 fk 开头

12. 唯一索引以 ux_ 为前缀

```
select constraint_name,table_name
from user_constraints
where constraint_type = 'U'
and substr(constraint_name, 1, 3) <> 'UX_'
and table_name not like 'BIN$%';
```

脚本 11-40 查询唯一索引是否以 ux 开头

13. 普通索引以 idx_ 为前缀

```
select index_name,table_name
from user_indexes
where index_type='NORMAL'
and uniqueness='NONUNIQUE'
and substr(index_name, 1, 4) <> 'IDX_'
and table_name not like 'BIN$%';
```

脚本 11-41 查询普通索引是否以 idx 开头

14. 位图索引以 bx_ 为前缀

```
select index_name,table_name
from user_indexes
where index_type LIKE '%BIT%'
and substr(index_name, 1, 3) <> 'BX_'
and table_name notlike 'BIN$%';
```

脚本 11-42 查询位图索引是否以 bx 开头

15. 函数索引以 fx_ 为前缀

```
select index_name,table_name
from user_indexes
where index_type='FUNCTION-BASED NORMAL'
and substr(index_name, 1, 3) <> 'FX_'
and table_name notlike 'BIN$%';
```

脚本 11-43 查询函数索引是否以 fx 开头